FORMAL VERIFICATION OF A NETWORK ON CHIP

A thesis

Submitted to the African University of Science and Technology

Abuja – Nigeria

in partial fulfillment of the requirement for the degree of:

MASTER OF SCIENCE IN COMPUTER SCIENCE

By

BEIDU, SANDY KWAKU

40030

Supervisor:

Prof. Jeff Sanders

December 2009

FORMAL VERIFICATION OF A NETWORK ON CHIP

by

Sandy Kwaku Beidu

RECOMMENDED : .....

.....

Committee Chair

APPROVED: .....

Chief Academic Officer

.....

Date

Abstract

Current advancement in VLSI technology allows more circuit to be integrated on a single chip forming a System on Chip (SoC). The state of art in on-chip intermodule connection of using a shared bus with a common arbiter poses scalability problems and become a performance bottleneck as the number of modules increase. Network on Chip (NoC) has been proposed as a viable solution to this problem. The possibility of occurrence of deadlocks and livelocks in a NoC requires that their design be validated since these can cause serious consequences such as power consumption and heat dissipation.

The traditional ways of validating chips by simulation-based techniques are been stretched passed their limits and the only alternative left is formal verification. This project pushes forward the range of applicability of formal verification by formally verifying the OASIS NoC using the model checking technique. Both refinement model checking and probabilistic model checking techniques are used to verify OASIS NoC for properties of the System. The OASIS NoC is first formalised in CSP and then verified with the FDR model checker for deadlock freedom. It is also shown that PRISM model checker which is designed for verifying probabilistic properties can be used to verify non probabilistic properties by using PRISM to also verify the OASIS NoC for deadlock freedom. The verification result of both FDR and PRISM shows that the OASIS NoC is free from deadlock. It was also shown using PRISM that the OASIS NoC behaves as a message buffer as expected of NoCs.

Acknowledgements

To God be all the glory for the great things he has done in my life.

I would like to express my sincere gratitude to my supervisor, Prof. Jeff. Sanders, for his guidance, support and encouragement that saw me through this work successfully.

I am also grateful to Prof. A. B. Abdallah, for introducing me to Network on Chips and also allowing me to use his work for this case study.

I would also want to appreciate Dr. K. Voltaire and Dr. B. Cisse for their support in helping me attend SEFM ’08 which introduced me to formal methods.

Many thanks to my family and friends for their love and care that has brought me this far.

Finally I would like to thank all members of the AUST FWC Care Group including the coordinators for their prayers and encouragement.

Table of contents

Abstract ..... **i**

Acknowledgements ..... **ii**

Table of contents ..... **iii**

List of Figures ..... **vi**

Chapter 1 - Introduction **1**

1.1 OVERVIEW 2

1.2 RELATED WORK 3

1.3 REPORT STRUCTURE 4

Chapter 2 – Network on Chip **5**

2.1 TOPOLOGY 6

2.1.1 Regular topologies 8

2.1.2 Irregular topologies 10

2.2 ROUTING 11

2.3 SWITCHING 13

2.4 FLOW CONTROL 15

2.5 BUFFERING 17

2.6 OASIS NOC 18

2.6.1 Topology 18

2.6.2 Switching and Routing 19

2.6.3 Flow control 22

Chapter 3 – Formal Verification **24**

3.1 INTRODUCTION 24

3.2 MODEL CHECKING 26

3.2.1 Modelling a system 28

3.2.2 Specification of system Properties 29
3.2.3 Temporal Model checking problem 31
3.2.4 State Space Explosion 32

Chapter 4 – Refinement Model Checking 35

4.1 CSP 35
4.1.1 The CSP Language 36
4.1.2 Observing Process behaviour – Failure, Divergence and Refinement 45
4.1.3 Operational Semantics 49
4.2 FDR 52
4.2.1 Verification with FDR 53
4.3 ProBe 56

Chapter 5 – Probabilistic Model Checking 58

5.1 Introduction 58
5.1.1 Probabilistic model checking 58
5.1.2 Overview of PCTL 59
5.2 PRISM language 60
5.3 PRISM Properties specification 63
5.3.1 The P operator 64
5.3.2 The S operator 65
5.3.3 Rewards-based properties - R operator 65
5.3.4 Quantitative properties 66

Chapter 6 - Model Checking OASIS with FDR 68

6.1 Modeling OASIS in CSP 68
6.1.1 Model Parameters and channels 69
6.1.2 Input Buffer model 70
6.1.3 Route model 71
6.1.4 Router model 73

6.1.5 Network model 73
6.2 Verification in FDR 74
6.3 FDR Verification Results and Analysis 74

Chapter 7 – Model Checking OASIS with PRISM 76

7.1 Prism model of OASIS 76
7.2 PRISM Verification of OASIS – Results and Analysis 79
7.2.1 Verification of deadlock-freedom 79
7.2.2 Buffer property verification 81

Chapter 8 - Conclusion 84

Appendix A – PRISM Verification Results 86

References 89

List of Figures

Figure 2-1 Bus based communication of SoC 5
Figure 2-2 NoC based architecture 6
Figure 2-3 NoC components 7
Figure 2-4 Seperating communication from computation 8
Figure 2-5 NoC Ring topology 8
Figure 2-6 Noc Mesh topology 9
Figure 2-7 Noc Torus topology 10
Figure 2-8 Noc Fat-Tree topology 10
Figure 2-9 OASIS Noc topology 18
Figure 2-10 oasis crossbar 19
Figure 2-11 One OASIS router data path 20
Figure 2-12 OASIS flit structure 20
Figure 2-13 OASIS input port 22
Figure 2-14 OASIS Stall-go flow control 23
Figure 3-1 Model checking 27
Figure 3-2 Model checking problem 32
Figure 4-1 Counter-example in FDR 55
Figure 4-2 Using ProBE to animate a process 57
Figure 6-1 OASIS NoC CSP model digram 69
Figure 6-2 FDR Deadlock check of OASIS NOC 75
Figure 7-1 Diagram of PRISM module of Router_00 77
Figure 7-2 Router_00 PRISM module 78
Figure 7-3 Path for a flit from ROUTER_01 to ROUTER_10 79

Chapter 1

Introduction

"Thus, there is a continual need to strive for balance, conciseness, and even elegance.
The approach we take, then, can be summarized in the following:
Use theory to provide insight; use common sense and intutition where it is suitable,
but fall back upon the formal theory when difficulties and complexities arise."
David Gries (The balance between formality and common sense, 1981)

Computers are increasingly becoming ubiquitous and the society's dependence on computers cannot be overemphasized. Imagine being in a future car which uses x-by-wire (where x is steer, break, gear, etc) technology and you apply the brakes but the car refuses to stop because the chip responsible for the breaking system is not responding. This failure may be due to the fact that the computerized breaking system which is likely to be a System on Chip (SoC), deadlocked and therefore no communication between the chips internal modules were possible. But fortunately, this is never going to hapen not because the current traditional simulation-based verification techiniques are enough to verify these chips but rather more robust formal techniques will be applied to ensure that such a situation is not possible by design.

The good news is that, although the communication structure of SoCs tend to have complex state machines, flow control logic and handshaking protocols, the design blocks that make up such networking chips are well-suited for formal verification. Specifying properties that describe proper operation of these chips, while not trivial, is a task that designers and verification engineers can accomplish with little training[Nar04].

In short the motivation for this work is: the society is increasingly becoming dependent on computers. Computers are becoming more and more complex and therefore we need to ensure these systems are dependable and increase people's confidence in using them.

1.1 Overview

As the number of modules integrated on a single die multiplies and VLSI technology scales toward deep sub-micron level, on-chip inter-module communication becomes a performance bottleneck. The state of art in on-chip module interconnection is a shared bus with a central arbiter. But as the number of modules increase, this common shared bus approach poses serious scalability problems which results in low performance and energy inefficiencies. Network on Chip (NoC) has being proposed as a viable solution to address the problem of on chip communication scalability issues.

NoC has gained a lot of attention in the SoC community and even among traditional data network communities. Already a lot of papers have been written on their design and implementations; some of which have been synthesized on FPGAs and on ASICs. Examples of such NoCs include the Æthereal, Nostrum, Spidergon, Octagon, XPipes and FAUST.

The NoC paradigm brings along it new research challenges. Due to their nature, NoC is restricted by low latency, area, power and heat dissipation. To reduce communication latency, various switching and routing techniques are employed some of which makes the NoC prone to deadlocks. Again, restriction on area puts a limit on router buffer capacity which overloads the network and thereby increases the possibility of deadlock. Livelocks can also cause serious consequences with respect to power consumption and heat dissipation. While conventional data networks, like the internet, can accept packet losses and resending of packets, this is unacceptable in a NoC environment. Because of these reasons it becomes extremely

necessary to verify their design. The size and complexity of SoCs is growing at a faster rate that traditional based techniques of simulation and testing are being stretched passed their limits. The only viable alternative is the use of formal verification methods whereby properties are proved to be true in a system by using mathematical techniques.

The goal of this project therefore is to push forward the range of applicability of formal methods by formally verifying a NoC using model checking techniques. The OASIS NoC which is a realization in FPGA of an earlier proposed Basic Network on Chip (BANC) by Abderazak and Sowa [AS06], will be used as a case study. Properties to be verified will be deadlock-freedom and whether the network behaves as a buffer without any flit losses. The FDR model checker will be used to verify the NoC for deadlock freedom. We also show how the PRISM model checker which is designed for verifying probabilistic properties can be used to very non-probabilistic properties by verifying OASIS NoC for deadlock freedom and also buffer property.

1.2 Related work

NoC is a more recent design paradigm and therefore little work has been done on their formal verification. One of the most notable works in this area is the work by Gebremichael et al.[GVZ+05] who verified the Philips Æthereal Protocol[GDR05] in PVS logic.

Schmaltz and Borrione also proposed a Generic Network on Chip framework (GeNoC)[SD05]. GeNoC is a metamodel of NoC which was developed and implemented in the ACL2 theorem prover. In a latter work Schmaltz et al. used their earlier work, GeNoC, to verify the Spidergon NoC from STMicroelectronics [SB07]. Also in [BHPS07], the Hermes [DHPS07] NoC was formalised in the GeNoC framework and was formally verified.

In [SSTV07], the CADP toolbox [GLM01] was used to perform a formal verification of an Asynchronous Network on Chip (ANoC) specified in the Communication Hardware Process(CHP).

1.3 Report structure

The rest of this report is organized as follows. The first part of Chapter 2 introduces the concept of Network on Chip describing properties of NoCs such as the topology, switching, routing and flow control techniques. The second part of Chapter 2 presents the OASIS NoC which is used as a case study for this work. In Chapter 3 we give an introduction to formal verification techniques with emphasis on the type of technique used in this work, namely model checking. In this chapter we provide a way of modelling a system and how to specify its properties. The chapter ends with a discussion on one of the main shortcomings of the model checking technique, the state space explosion. Chapter 4 presents a background on how to model a system and its properties using the CSP approach and then present the FDR tool which can be used to verify a system specified in CSP. Chapter 5 presents a different approach to model checking, called probabilistic model checking and presents the PRISM language and tool. The OASIS NoC is formalized in CSP and PRISM language and then verified for deadlock freedom using FDR and PRISM tool in Chapter 6 and Chapter 7 respectively.

Finally, Chapter 8 concludes this report, with a summary of this work and provides a discussion of possible future directions for this work.

Chapter 2

Network on Chip

Current state of the art in chip production allows more and more circuits to be integrated onto a single chip. This idea of having a complete system on a chip is termed System on Chip (SoC). Future SoC are predicted to become communication bound. Conventional SoC uses a shared bus for interchip module communication which does not scale well with increase in number of modules.

The Network on Chip (NoC) paradigm has been proposed as a viable solution to meet future SoC requirements.

FIGURE 0-1 Bus based communication of SoC

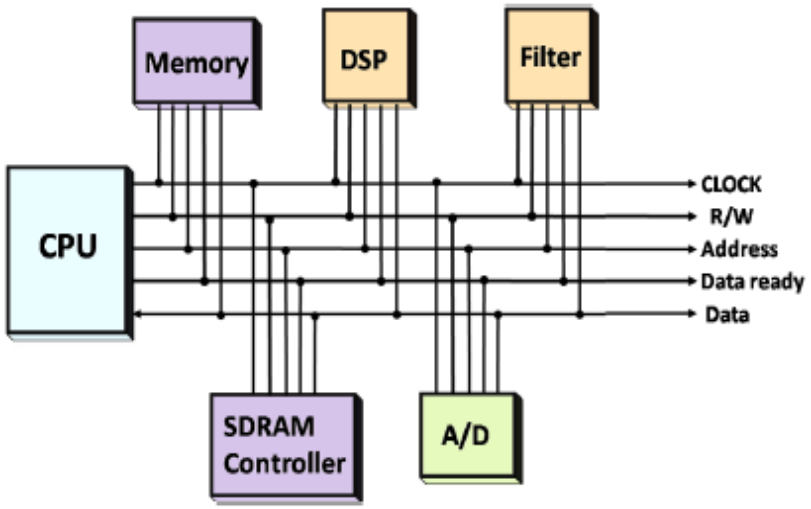
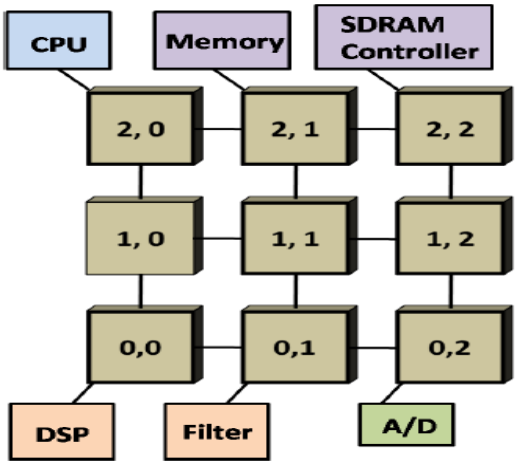


Figure 0-2 NoC based architecture



The idea behind the NoC concept is to replace point-to-point wires and global shared buses (Figure 0-1), with a generic communication medium (Figure 0-2) that can support all types of communication. NoC is a set of interconnected routers, with Intellectual Property (IP) cores connected to these routers. The routers are responsible for: (i) receiving incoming packets; (ii) storing packets; (iii) routing these packets to a given output port; (iv) sending packets to others routers.

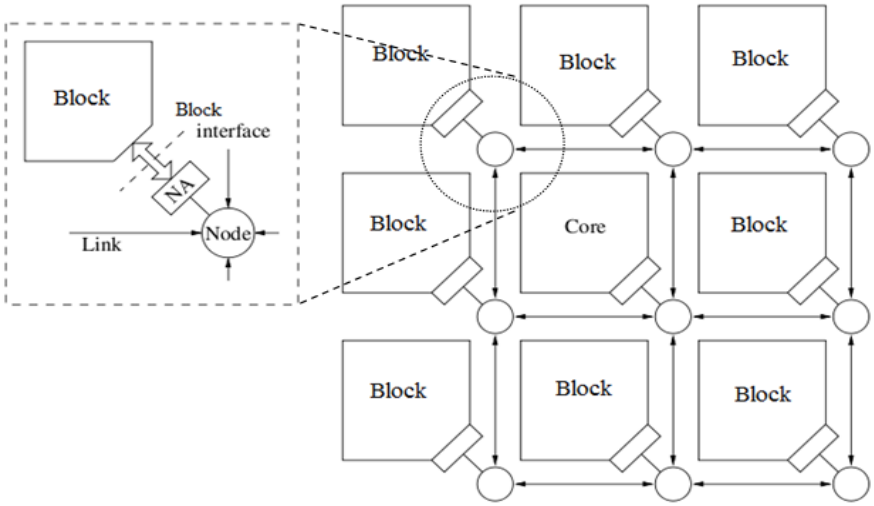
The first part of this chapter takes a look at some of the characteristics of NoCs and the second part describes the OASIS NoC which was used as a case study for this work.

A network on chip can be characterized by its topology, routing algorithm, switching techniques and flow control mechanisms [NM93].

2.1 Topology

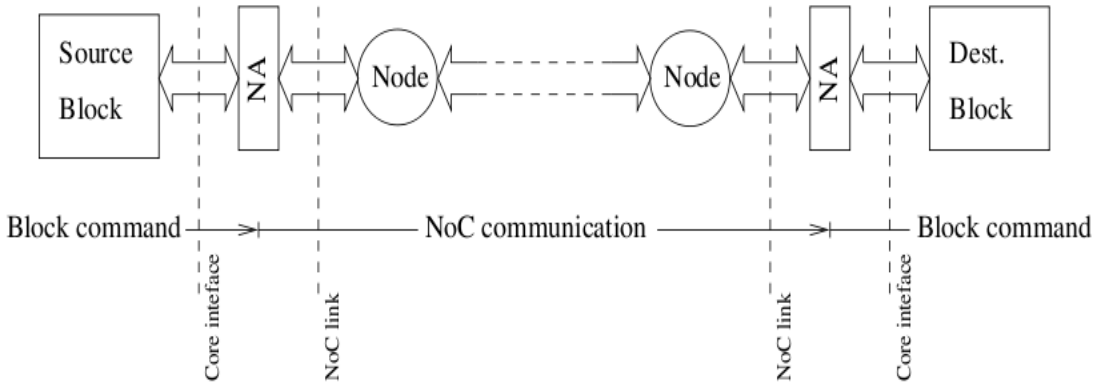
The topology of a NoC is the physical structure of the network graph. That is how the components of the network are connected together. A typical NoC consists of the following components: blocks, nodes, links (also referred to as channels), and Network Adapters (NA).

Figure 0-3 NoC components



- **Blocks** are normally the computational or other modules that need to communicate in the network. These include microprocessors, memories (e.g. RAM), graphic adaptors and special purpose IP cores (e.g. DSP, Ethernet, USB, etc).
- **Routing nodes** or simply nodes routes data through the network according to a routing strategy. In NoC routing nodes are normally implemented as routers.
- **Links** are used to connect routing nodes to each other allowing data to be transferred from node to node. Links may be uni-directional or bi-directional and provide the raw bandwidth in the network. They may contain one or more physical or logical channels.
- **Network Adapter** is the interface between the network and the blocks. It hides the implementation details of the network (communication structure) from the blocks (computation structure). The blocks do not need any knowledge about the network to communicate. This decoupling of computation from communication allows independent design and implementation and reuse.

Figure 0-4 Seperating communication from computation



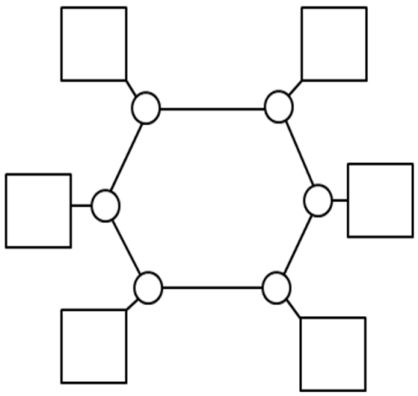
The topology of a network can be classified broadly as regular or irregular.

2.1.1 Regular topologies

Regular topologies have symmetric shapes and are often used for general networks and networks that can be parameterized. Examples of common regular NoC topologies are described below.

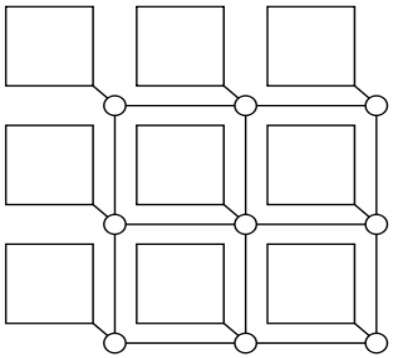
- **Ring**: The ring is the least complex topology where each node is connected to two other nodes to form a circular shape. Ring topology is highly regular, simplifying the routing scheme and flow control. Its disadvantage is that it scales poorly due to limited connectivity. Examples of NoCs employing this topology are Proteo [STN02], Octagon[KND02] and Ring Road[SK04].

Figure 0-5 NoC Ring topology



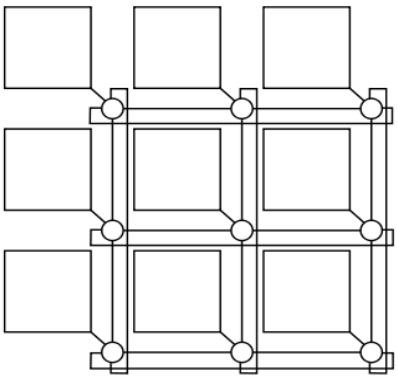
- **2-D mesh**: This is a two dimensional grid topology consisting of vertical and horizontal links with nodes connected at their intersection. It is the most simple 2-D network structure. This topology is the most favoured topology of NoCs so far because of its simplicity, regularity and linear area growth with the number of nodes. Also inter-node delay can be predicted with high accuracy. Meshes have a relatively long average network distance, which can increase power consumption. Examples of mesh-based NoCs include Nostrum[MNTJ04] and OASIS(adopted for this project).

Figure 0-6 Noc Mesh topology



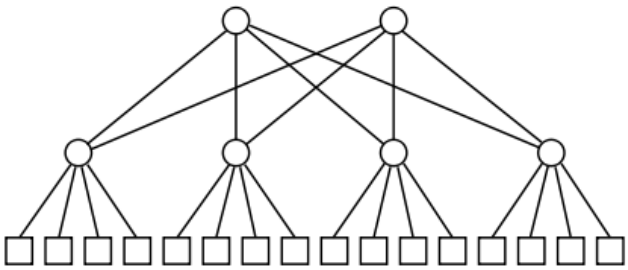
- **2-D torus**: this topology is similar to the 2-D mesh topology, but with opposite edge nodes connected together. The torus topology reduces the inter-node distance of edge nodes and also avoids routing problems at the edge of the nodes. An example of a torus NoC is Ahmad[AEK06]

Figure 0-7 Noc Torus topology



- **Fat-Tree**: The tree is a hierarchical topology that begins with a top-level root node with connections to one or more nodes (known as children) at a lower level. This connection pattern continues to establish a tree structure that has *d* levels, where each node has *k* children. The blocks are connected to the leaf nodes (lowest level). Examples of fat-tree based NoC are SPIN[GG00] and Butterfly FT [FP07]

Figure 0-8 Noc Fat-Tree topology



2.1.2 Irregular topologies

An irregular topology can be derived by altering the connectivity of a regular topology (like the regular ones discussed previously), to form hybrid, cluster-based, or asymmetric topologies.

Irregular topologies can be used to create more specific topologies that meet the communication requirements of the system better and reduce overhead capacity.

The disadvantages of irregular topologies are that they are often complex, requires special routing and control flow schemes and does not scale linearly in area and power. The Xpipes[BB04] and Æthereal[GDR05] NoCs support this type of topology.

2.2 Routing

The routing scheme or algorithm of a NoC determines the high-level path through which data is transferred from source to destination. There is a difference between routing and switching. While switching defines the low-level transport of data, routing is the intelligence behind this transport that determines the path through the network.

Routing algorithms must prevent *deadlock*, *livelock*, and *starvation* [LST00], NM93] situations. Deadlock may be defined as a cyclic dependency among nodes requiring access to a set of resources, so that no forward progress can be made, no matter what sequence of events happens. Livelock refers to packets circulating the network without ever making any progress towards their destination. Starvation happens when a packet in a buffer requests an output channel, being blocked because the output channel is always allocated to another packet.

Routing algorithms can be classified according to the three different criteria:

- how a path is defined;
- where the routing decisions are taken, and
- the path length.

In terms of how a path is defined, routing can be classified as *deterministic* or *adaptive routing*. In a deterministic routing all packets follow the same path for a given source-destination pair irrespective of the network status. That is, given a source and a destination, deterministic routing will choose the same path. A

common deterministic routing used in meshes and tori is the *XY scheme* which is simple and robust against deadlocks [DYN03]. XY is a tableless routing scheme whereby each packet is routed first in a horizontal(X) direction and then along the vertical(Y) direction. The Proteo[STN02], Mango[BM06], Xpipes[BB04] and Octagon[KND02] are all examples of NoCs that use a deterministic routing scheme. In an adaptive routing scheme, routing decisions are made on a per-hop basis at each routing node based on the current state of the network. The network state may include the status of a node or link, the length of queues, and historical network load information. Adaptive routing is used in the hope of reducing network congestion and balancing the network load. Compared to deterministic routing, adaptive routing is more complex to implement and it requires special attention to avoid deadlocks. Because of the high tendency of deadlock and livelock situations to happen in fully adaptive algorithms [8], its usage is limited in NoC designs. Examples of NoCs that use adaptive routing includes the Nostrum[MNTJ04] , SPIN[GG00] and Butterfly FT [FP07]. There also exist partial adaptive schemes that combine both deterministic and adaptive schemes. An example of such a scheme is the Toggle-XY(TXY) that adaptively chooses between deterministic XY or YX [SALR05].

According to where routing decisions are taken, the routing algorithm can be classified as *source* or *distributed* routing. Source routing derives its routing decision from a pre-computed global routing table, such that all the necessary routing decisions are embedded in the header to guide a packet to the receiving node. Since the packet header has to carry all the routing information, the packet size is increased. Distributed routing derives routing decisions locally at each router from local lookup tables or hardware routing functions. Hence, a packet only carries the destination address in its header, reducing the header overhead in a packet. Distributed routing can also take into account faulty paths, resulting in fault tolerant algorithms.

Regarding the path length criterion, routing can be *minimal* or *non-minimal* [LST00] [NM93]. A routing algorithm is termed minimal if it always chooses the shortest path between source and destination otherwise it is non-minimal routing scheme. An example of a minimal routing algorithm is the XY. In non-minimal routing, the packet can follow any available path between source and target. Nonminimal routing offers great flexibility in terms of possible paths, but can lead to livelock situations and increase the latency to deliver the packet.

2.3 Switching

The switching scheme describes how data moves through a switch in a router. The switch connects the input ports to the output ports. The two basic switching techniques used in NoCs are *circuit switching* and *packet switching*.

In circuit switching, prior to transmission, a dedicated end-to-end path (both routers and links) is reserved from the source to destination. This path is maintained for as long as the duration of the transmission. One of the main advantages of using circuit switching in NoCs is that because buffers are not needed, there is a reduction in area and power which are critical in NoC designs. Although circuit switching can potentially ensure full utilization of the available bandwidth when the volume of communication is high, there is a high initial latency associated with the set-up of the dedicated circuit. Circuit switching is therefore best suitable for application-specific SoCs whereby the communication patterns and transfers are long enough to deserve the setup latency. Examples of NoCs that use circuit switching scheme include CrossRoads[CSC06], PNoC[HN05], Wolkotte[W05] and Nexus[Lin04].

Packet switching on the other hand, transfers data by segmenting it into smaller units called packets and then forward individual packets from the sender to the receiver on a per-router basis. Depending on the routing scheme, packets may follow different routes and may also have different delays. A typical packet consists of a

control part and a data part (also referred to as payload). The control part is used by routing nodes to determine which route a packet should take. Packet switching requires buffering and therefore introduces unpredictable latency in the transmission. Nevertheless it is more flexible especially for small transfers and offers higher link utilization for general and unknown communication patterns. A majority of NoCs proposed or implemented in FPGA uses packet switching scheme. In a survey of network on chip proposals, Salminen et al.[SKH08] found 80% of the studied NoCs to utilize packet switching.

How the packets are stored passed between the routing nodes is known as *forwarding scheme*. The basic forwarding schemes used in packet switching are Store-and-forward (SAF), Virtual-cut-through (VCT) and wormhole (WH) switching[DYN03].

- **Store-and-forward (SAF)** switching interprets the header information at each intermediate router and uses the information to determine the output link through which the packet is to be forwarded. A packet transfer is initiated only when the receiving router has sufficient buffer space for the entire packet, which requires the buffer size to be at least the size of a packet. A stalled packet occupies the local node itself.

- **Virtual-cut-through (VCT)** switching is similar to SAF switching, but it does not wait for a packet to be received in its entirety before making routing decisions. Transfer latency can be reduced by interpreting the header as soon as it is available, without waiting for the data payload to be received after the header. The packet is forwarded to the next router only when there is available buffer space for the entire packet, otherwise the packet is buffered at the local node. The buffer requirement and the cost for stalling are the same as SAF switching.

- **Wormhole (WH)** switching[NM93] reduces the buffer requirement in each router by dividing packets into smaller segments called flits (flow control units) and pipelining them through the network. The header flit is interpreted and immediately forwarded when there is space for that flit in the receiving router. The remaining flits of the same packet are forwarded in the same way as the header as they arrive. As a result, a packet occupies buffers in several routers and the links between them.

2.4 Flow Control

A network consists of many links (channels) and buffers. Flow control deals with the allocation of the links and buffers to a packet as it travels along a path through the network. A resource collision occurs when a packet cannot proceed because some resource that it requires is held by another packet. Whether the packet is dropped, blocked in place, buffered or re-routed depends on the flow control policy. A good flow control policy should avoid channel congestion while reducing the network latency[NM93].

Store-and-forward switching and virtual cut-through switching allocate channel bandwidth and buffers in units of packets while wormhole switching allocates both channel bandwidth and buffers in units of flits.

There are two very important issues to consider in the allocation of channels and their associated buffers to packets (or flits). Since the routing algorithm determines which output channel is selected for a packet arriving on a given input channel, the routing algorithm can be referred to as the *output selection policy*. On the other hand, a particular output channel can be requested by packets arriving on many different input channels. Therefore there should be a means of determining which packet may use the output channel. This is referred to as the *input selection policy* or *arbitration*. The input selection policy chosen will affect the fairness of the routing

algorithm. The arbitration policy should ensure that no packet experiences starvation. Examples of such input selection policies include *first-come-first-serve*, *round robin* and *channel priority*.

Flow control can be implemented at a high level, known as end-to-end flow control, in order to manage the flow of information for the entire path between a sender and receiver. Flow control can also be implemented at a lower level, such as the link level, where the control of information flow occurs between routers. Most NoC's flow control schemes are implemented at the link level because less complexity is required to manage the flow of information locally between two routers.

Many link level flow control mechanisms have been proposed for NoCs. Some of these schemes are:

- **Credit-based** flow control: With credit-based flow control, the upstream switch keeps track of the number of free flit buffers in each buffer queue downstream. If the upstream switch sends a flit, it decrements the corresponding count. If the downstream switch forwards a flit, it sends back a credit to the upstream switch, causing a buffer count to be incremented **[BCGK04]**.

- **Ack / Nack** flow control: With this scheme there is no state kept in the upstream switch to represent buffer availability. The upstream switch optimistically sends flits to the downstream switch. If the downstream switch has a free buffer, it accepts the flit and sends an acknowledge (*ack*) to the upstream switch. Otherwise, the downstream switch drops the flit and sends back a negative acknowledge (*nack*). The upstream switch must hold each flit until it receives an *ack*. If it receives a *nack*, it retransmits the flit.

- **Stall-go** flow control: In this scheme each downstream router signals the

upstream routers whether its input buffer is full and therefore no transmission should proceed or the buffer is not full and therefore it is ready to receive. When the downstream router's input buffers are full packets are not transmitted, hence there are no packet drops in this scheme. This scheme is favored by many NoC designs [MCM+04] for its low complexity, good performance, and the ability to provide reliable data transmission.

2.5 Buffering

NoCs use buffering to provide temporary storage of packets (flits) that are in transit in routing nodes. Most NoCs implement buffering with FIFO memory, where packets are queued in a first come first serve basis. Buffering is an import issue to consider in the design of NoCs because of area and power requirements. There is a tradeoff between seeking to minimize the amount of buffering while still maintaining the specified performance.

There are various buffering schemes, namely centralized, input, and output buffering. Tamir and Frazier[TF88] provided an overview of the advantages and disadvantages for each buffering configuration. With respect to input and output buffering, they have shown that the mean queue length of systems with output buffering is always found to be shorter than the mean queue length of an equivalent system with input buffering. This is because with output buffering, switching is performed before buffering, hence inputs are isolated from output congestion up to the point that a target output has no remaining buffer capacity. Centralized buffering shares a set of buffers between multiple input and output ports. It is costly to implement, however, because of the associated control overhead. The concept of middle buffering has been proposed by Zimmer et al.[ZZHG05], which places buffering within the crossbar switch to provide smaller designs and superior performance than output buffering.

2.6 OASIS NOC

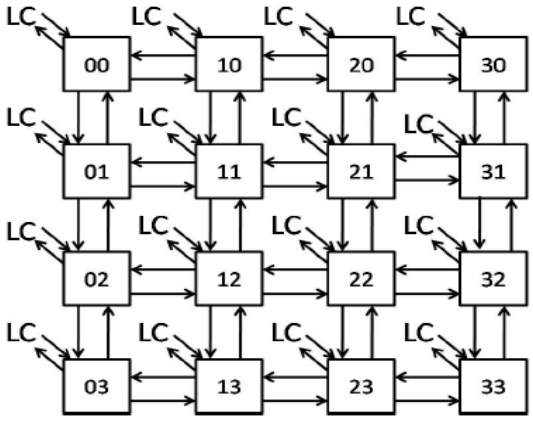
This project is a case study of an existing NoC and therefore an introduction to the NoC is necessary to gain enough understanding of the design and workings of it in order appreciate this work. The rest of this chapter is dedicated in describing the OASIS NoC which was used as a case study for this thesis work.

The OASIS NoC[MAK09] is a complexity effective on-chip interconnection network designed and prototyped on a FPGA. The aim of the OASIS NoC design was simplicity and a small hardware footprint. The OASIS NoC is actually an enhancement of an earlier work of Abdallah and Sowa, BANC [AS06]. It is adopted as the use case for this thesis work. Here we present the architecture of the OASIS NoC.

2.6.1 Topology

The OASIS NoC is a 2-D 4×4 mesh topology. Each routing node has an X-Y coordinate which is called its address. The number of input and output ports of each routing node ranges from 3-5 depending on the location of the routing node on the grid. Routing nodes located at the four corners have three ports, those located at the edges have four ports and the rest each has five ports. These ports are labeled *Local*, *North*, *South*, *East* and *West*. The local ports are used for inputting and outputting flits to and fro the network respectively.

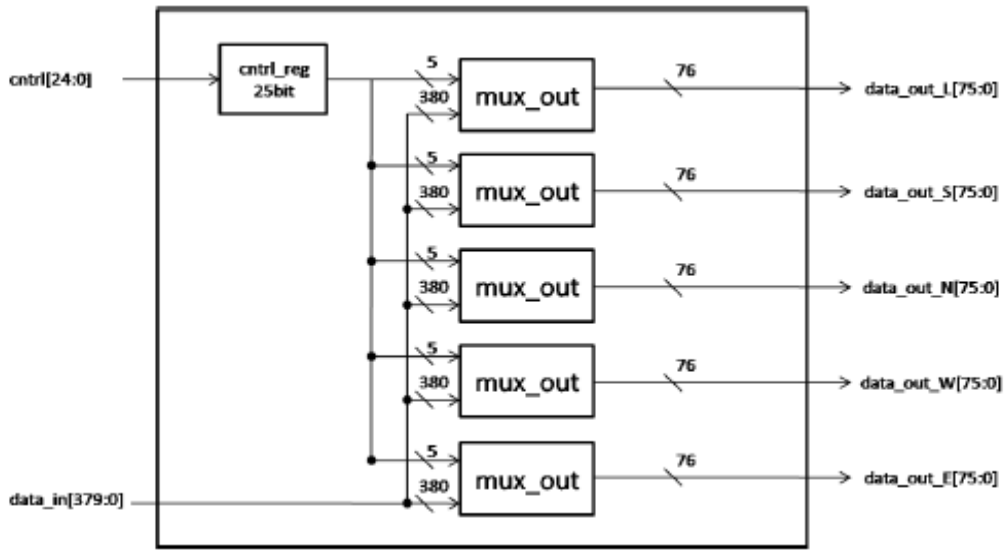
Figure 0-9 OASIS Noc topology



2.6.2 Switching and Routing

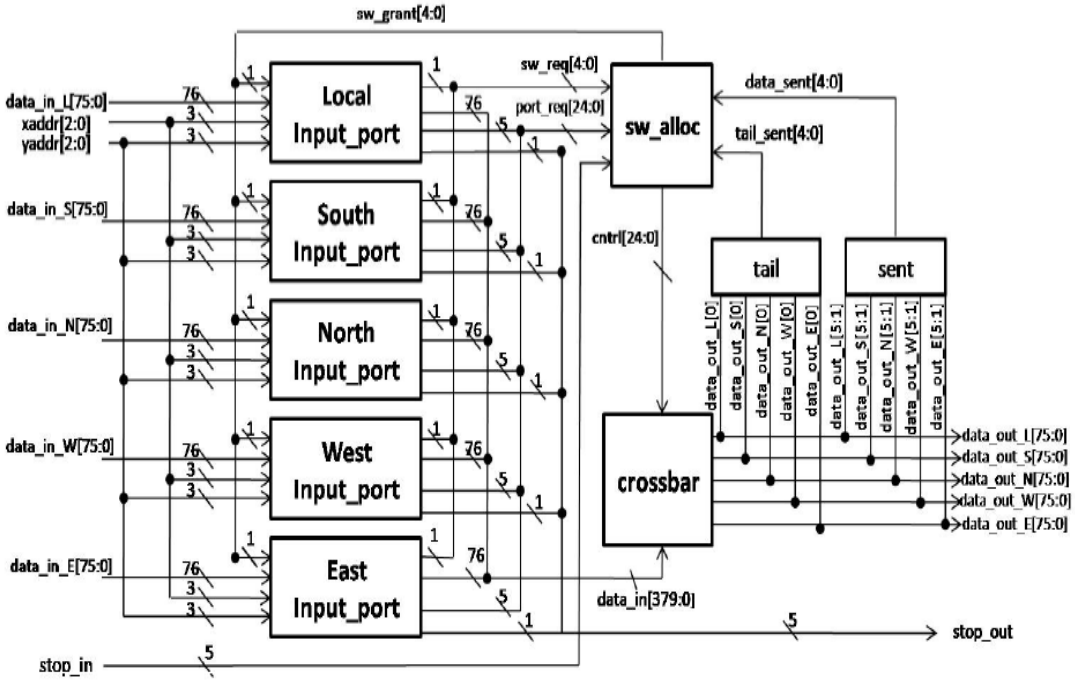
OASIS uses packet switching with packets divided into smaller flits. Switching in OASIS is performed by two main components, the **crossbar** and the **switch allocator (sw_alloc)**. The crossbar is a circuit that fully connects all input ports to output ports. The crossbar allows five different data to be routed at the same time.

FIGURE 0-10 OASIS CROSSBAR



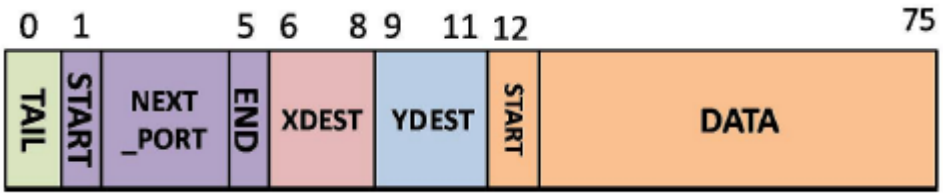
A very important point to note about the OASIS wormhole scheme is that, while in the conventional wormhole switching, control flow information is inserted in the header flit and the rest of the flits follow the path reserved by the header flit, each OASIS flit contains its own control information. This means that each flit is routed on its own. So one can say OASIS does not actually implement a full wormhole routing scheme.

Figure 0-11 One OASIS router data path



OASIS flit consists of 76 bits with a structure shown in Figure. The first field, **tail**, is a 1 bit used to determine if the flit is the last flit of the packet or not. This is needed by the Network Adapter when assembling flits to form packets. OASIS uses a *look-ahead routing* and therefore each flit contains the address of the **Next_Port** it is heading towards to which can be **L**, **N**, **S**, **E** or **W**.

Figure 0-12 OASIS flit structure



The **XDEST** and **YDEST** are the X and Y coordinates of the destination router. The 64 bit **DATA** field is the payload which contains the actual data being transmitted.

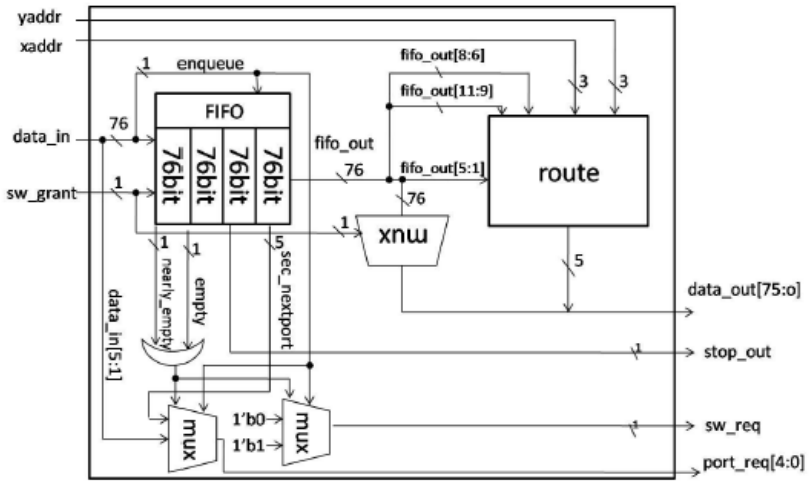
OASIS employs a *deterministic X-Y routing* scheme whereby all flits follow the same path for any given source-destination pair of addresses. X-Y routing is a shortest path algorithm. Routing decisions are taken in each routing node without using any lookup tables. When a router receives a flit in its input port, the next port of the flit is determined in the input port by the **Route** module. The **Route** module first determines the next address by using the **Next_Port** of the flit. The algorithm for doing this is as shown below:

```
//for next X address
if (next_port == EAST)
        next_Xaddr = Xaddr + 1
else if (next_port == WEST)
        next_Xaddr = Xaddr - 1
else next_Xaddr = Xaddr
//for next Y address
 if (next_port == NORTH)
        next_yaddr = yaddr + 1
 else if (next_port == SOUTH)
        next_yaddr = yaddr - 1
 else next_yaddr = yaddr;
```

After determining the x and y address of the flit, these are used also to calculate the new next port of the flit as follow:

```
 if (next_Xaddr = Xdest)
        if (next_Yaddr = Ydest)
                output = SELF
        else if (next_Yaddr < Ydest)
                output = NORTH
        else
                output = SOUTH
else
   if (next_Xaddr < Xdest)
        output = EAST
     else
        output = WEST
```

Figure 0-13 OASIS input port

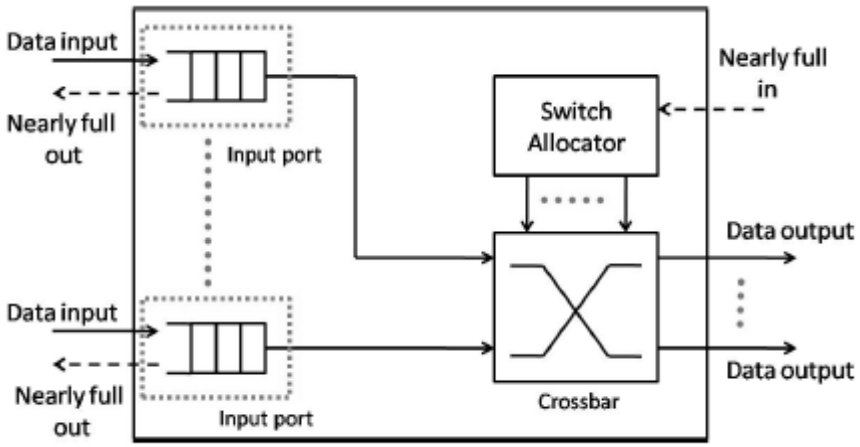


This information is then sent to into the **switch allocator** which determines when the flit should be sent. There may more than one flit in the input ports which may be requesting the same output port. The decision as to which of the input ports is allowed to send to the output port is determined by the **arbiter** which is found in the **switch allocator**. OASIS **arbiter** uses *round-robin scheme* with no priority for making such decisions. After the decision has been done this is sent as control information to the crossbar which then connects the particular input port selected to the output port for transmitting.

2.6.3 Flow control

Each input port of the OASIS router contains a **FIFO buffer** which can store four flits at a time. Because of the fixed size of the FIFO buffer in the input ports of the routers, input flits will overflow if the FIFO buffer is full. The **Stall-Go** flow control is used to avoid such a situation. This is one of the main differences between the OASIS NoC and its predecessor BANC[AS06] which was using a **Drop-tail** flow control whereby each flit finding the input port full is dropped and resent at another time. Before any router transmits any flit to a neighbouring router, it has to check whether the receiving input port buffer is not full.

Figure 0-14 OASIS Stall-go flow control



Chapter 3

FORMAL VERIFICATION

3.1 Introduction

We say a system is correct if it meets its designed requirements. There is no notion of an "absolute" correct system; correctness is only meaningful with respect to a given specification. For instance we cannot say this NoC is correct; rather we can say this NoC is correct with respect to deadlock-freedom. To ensure the reliability of a system we should be able to show that a system is correct with respect to its specification. It is not enough to just show that the system *can* meet its requirement but rather we should also be able to show that the system *cannot fail* to meet its requirement. We can guarantee the correctness of systems with respect to requirements through system *validation*. Generally validation is the process of checking that something meets a specified criterion. The main validation techniques used in industry today are *testing*, *simulation* and *verification*. Although some authors like to talk about *Validation and Verification*(V&V) as complementary techniques, here we consider verification as a validation technique. We clarify the difference between the two terms with the following definitions:

- **Validation**: Are we building the right system? That is, does the system do what the user really requires?

- **Verification**: Are we building the system right? That is, does the system conform to its specification?

In the simulation technique, a model (abstract representation) of the system is created (usually in some programming language, e.g. Java) and different inputs are

run on the model and the output observed and analyzed to find out if it behaves as expected. One way of generating such inputs is by using random generators. For example Abderazek et. al. [AS06] in validating their buffer design of the BANC NoC used the Network Simulator(NS) from Berkeley[FV03]. They used random number generation (RNG) to generate X and Y coordinates for each source and destination as input to the simulator. The simulation technique for validating a system is not exhaustive.

Testing on the other hand checks the actual system rather than a model of it, usually by informal approaches (though there are some formal approaches). Testing focuses on sampling executions according to some coverage criteria which makes this approach non-exhaustive.

The last techniques, formal verification can be described as a process of applying a manual or automatic *formal* technique for establishing whether a given model of a system satisfies a given property or behaves in accordance to some abstract description (specification) of the system [Pel01]. Formal means the methods used are based on mathematical theories, such as logic, automata, graph, set theory or some algebra such as process algebra (used in CSP). Formal verification is under a bigger area, *formal method*. Formal methods can be described as using mathematically-based languages, techniques, and tools for specifying systems. The advantage of using formal methods is that it increases our understanding of the system by revealing inconsistencies, ambiguities and incompleteness that might otherwise go undetected, thereby increasing our confidence in the system.

The two main formal method techniques are theorem proving (deductive verification) and model checking (algorithmic verification).

Theorem Proving: Theorem proving is a technique where both the system and its desired properties are expressed as formulas in some mathematical logic. This logic is given by a formal system, which defines a set of axioms and a set of inference

rules. Theorem proving is the process of finding a proof of a property from the axioms of the system. Steps in the proof appeal to the axioms and rules, and possibly derived definitions and intermediate lemmas [CW06]. The major advantage of theorem proving over its counterpart model checking is that it does not suffer from state space explosion. Theorem proving can deal directly with infinite state spaces. The disadvantage of theorem proving technique is that the process is not fully automatic and therefore requires human expert interaction. This makes the technique slow and prone to errors. However in the process of finding the proof, the human user often gains invaluable insight into the system or the property being proved. Notable examples of theorem provers are Isabelle[Pau94] and PVS [ORS92].

Model checking which is the technique used in this work is discussed in more detail in the rest of this chapter.

3.2 Model Checking

Model checking is a technique that relies on building a finite model of a system and checking that a desired property holds in that model. The check is performed as an exhaustive state space search which is guaranteed to terminate since the model is finite. Two main advantages of model checking over theorem proving are:

i. It is fully automatic; therefore it requires no user interaction or human expertise.
ii. It produces counterexample when some property fails to be checked in the process. This is an invaluable piece of information for better understanding of the system and its possible unforeseen flaws.

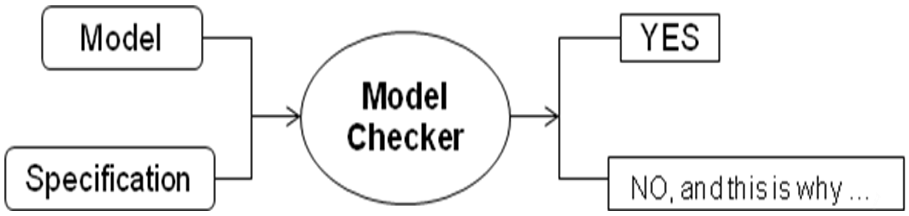
The main disadvantage of model checking is that it cannot handle large state space; the *state space explosion* problem. Various techniques have been proposed and implemented to address this issue. Some of these techniques for reducing the problem of state space explosion will be discussed in more details in a latter section in this chapter.

In generally, model checking consists of the following tasks:

1. Modelling the system; (this may require the use of abstraction)
2. Specifying the properties the design must satisfy
3. Verifying that the system satisfies its specification

A model checker is a tool used for performing model checking. A model checker takes in the model of the system and a property specification and then uses a certain model checking algorithm to verify whether the system satisfies its property. In the event that the verification fails, a counterexample is produced. Figure 3.1 illustrates the model checking technique.

Figure 0-1 Model checking



There are two main model checking techniques used in practice today. In the first approach, *temporal model checking* [CE81;QS82], specifications are expressed in a temporal logic such as LTL[Pnu81] and system is modelled as finite state transition system. An efficient search procedure is used to check if a given finite state transition system is a model for the specification. An example of a model checker that uses this technique is the PRISM model checker[PRIS] which is used in this work to verify some properties of a NoC.

In the second approach, *conformance model checking*, the specification is given as an automaton; then the system, also modelled as an automaton, is compared to the specification to determine whether or not its behaviour *conforms* to that of the

specification. Different notions of conformance have been explored, one of which is refinement orderings [CMS95, Ros94]. The Formal Systems' FDR model checker that will also be used in this work to verify deadlock-freedom uses this approach.

3.2.1 Modelling a system

The behaviour of a system can be modelled as a graph with nodes representing states of the system and edges representing state transitions. Information can be put on either the nodes or the edges leading to two different kinds of models. When the information is put on states, called a*tomic propositions*, the model is called a *Kripke structure*. On the other hand if the information is put on the edges, called *action labels*, the model is referred to as *Labelled Transition System (LTS)*. Below are the formal descriptions of the two models.

Kripke Structures: Let *AP* be a set of atomic propositions

A Kripke structure [CGP99] over AP is a structure

M = (S, S_0, R, L),

where

- *S* is a finite set of states
- $S_0 \in S$ is the set of initial states. Sometimes S0 is irrelevant and dropped; other times it is a single state.
- $R \subseteq S \times S$ is a total binary relation on *S*, so for all $s \in S, \exists\, t \in S$, such that $s\ R\ t$. *R* represents the set of transitions.
- $L : S \rightarrow 2^{AP}$, labelling function that labels states with atomic propositions

A path π in M is an infinite sequence s_0 , s_1 , s_2,⋯, such that for all *i*, $s_i \in S$ and $s_i\ R\ s_{i+1}$.

Labelled Transition Systems(LTS): A LTS is a five tuple $(S, S_0, A, \tau, \rightarrow)$, where

- *S* is a set of states

- $S_0 \in S$ denotes the initial state
- $\tau \in A$ denotes a hidden (silent, internal, invisible) step
- *A* is a set of Actions or Events
- $\rightarrow \subseteq S \times A \times S$ is the transition relation. We write $x \xrightarrow{a} y$

A convenience notation is $x \overset{a}{\Rightarrow} y$, which is defined as $x \xrightarrow{a} y \vee (a = \tau \wedge x = y)$. (reflexive closure for hidden steps)

3.2.2 Specification of system Properties

In the previous sections, we defined the Kripke structure and LTS in order to describe a system in a formal way. A temporal logic is then used to provide specifications (also called properties) for those systems and to prove them. Linear-time temporal logic(LTL) [CGP99, HR04], is a temporal logic to reason about states on future paths. An LTL formula ϕ is satisfied in a states, if all paths starting in that state satisfy ϕ. Thus, LTL implicitly quantifies universally over paths.

LTL has the following syntax given in Backus-Naur-form:

$$\varphi ::= \mathrm{T} \mid \mathrm{F} \mid \mathrm{p} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \mid \mathrm{X}\,\varphi \mid \mathrm{F}\,\varphi \mid \mathrm{G}\,\varphi \mid \varphi\,\mathrm{U}\,\varphi \mid \varphi\,\mathrm{R}\,\varphi$$

where *p* is any proposition atom from some set *A*. Thus, the symbols T(True) and F(False) are LTL formulas, as are all atoms from *A*, and ¬*φ* is an LTL formula if *φ* is one, etc. The connectives *X*, *F*, *G*, *U* and *R* are called temporal connectives. X means "ne**X**t state", F means "some **F**eature state", and G means "**G**lobally", i.e. all feature states. The connectives *U* and *R* are called "**U**ntil" and "**R**elease" respectively. Usually, *A* defines a set of atomic formulas denoting atomic facts which may hold in a system. An atomic formula $p \in A$ is then true in a state *s*, if the action *p* can be executed in this state. This brings us to the semantics of LTL.

A model for an LTL formula is defined by a transition system $M = (S, \rightarrow, L)$, where S is a set of states, $\rightarrow \subseteq S \times S$ is a binary relation on *S*, and $L : S \mapsto P(A)$ is a labelling function for states. The latter one defines the atomic formulas which are true in the

states. To reason about the future of a state $s \in S$, we need the definition of a path in a transition system M.

A path is an infinite sequence of states $s_1, s_2, s_3, \cdots$ in S such that, for each $i \geq 1$, $s_i \rightarrow s_{i+1}$ holds. We write paths as $\pi = s_1 \rightarrow s_2 \rightarrow \cdots$, The suffix of a path π, starting at state s_i , is denoted by π^i . Whether a path $\pi = s_1 \rightarrow \cdots$, of a transition system M satisfies an LTL formula is defined by the satisfaction relation $\vDash$ as follows:

1. $\pi \vDash T$
2. $\pi \vDash F$
3. $\pi \vDash p$, iff $p \in L(s_1)$
4. $\pi \vDash \neg\varphi$, iff $\pi \vDash \varphi$
5. $\pi \vDash \varphi_1 \wedge \varphi_2$, iff $\pi \vDash \varphi_1$ and $\pi \vDash \varphi_2$
6. $\pi \vDash \varphi_1 \vee \varphi_2$, iff $\pi \vDash \varphi_1$ or $\pi \vDash \varphi_2$
7. $\pi \vDash \varphi_1 \rightarrow \varphi_2$, iff $\pi \vDash \varphi_2$ whenever $\pi \vDash \varphi_1$
8. $\pi \vDash X\,\varphi$, iff $\pi_2 \vDash \varphi$
9. $\pi \vDash G\,\varphi$, iff $\forall i \geq 1.\pi_i \vDash \varphi$
10. $\pi \vDash F\,\varphi$, iff $\exists i \geq 1.\pi_i \vDash \varphi$
11. $\pi \vDash \varphi\, U\, \psi$, iff $\exists i \geq 1.(\pi_i \vDash \psi \wedge \forall j.1 \leq j < i \rightarrow \pi_j \vDash \varphi)$
12. $\pi \vDash \varphi\, R\, \psi$, iff $\exists i \geq 1.\ (\pi_i \vDash \varphi \wedge \forall j.1 \leq j \leq i \rightarrow \pi_j \vDash \psi) \vee \forall k \geq 1.\ \pi_k \vDash \psi$

For a given transition system $M = (S, S_0, R, L)$, a state $s \in S$ fulfils an LTL formula φ if and only if φ is satisfied by every path π in M starting at s.

We then write $< M, s > \vDash \varphi$.

Temporal logics allow some very useful and important properties of systems to be specified and verified with model checkers. Such properties include:

- **Safety properties**: This is where we are interested in verifying that "nothing *bad ever happens*". For example in a 2×2 2D-mesh NoC we may desire that it is not the case that "*all input buffers of all four routers get full*". We can specify this property in LTL as :

  $\boldsymbol{G}\ \neg(router00\_full \Longrightarrow router01\_full \wedge router10\_full \wedge\ router11\_full\ )$

- **Liveness properties**: If we desire the property that "*something good will happen*", we specify this as a liveness property. As an example using our previous NoC example, we can specify the property "*if a message enters router00, eventually it will come out of one of its two possible outputs*" as the LTL formula:

  $\boldsymbol{G}\ (enter\_router00 \Longrightarrow \boldsymbol{F}\ (\ out\_south\ \vee out\_east\ ))$

- **Fairness properties**: These are used to specify the properties that "*something will happen infinitely often*". Using our NoC example again, we can write the specification that "*messages will be input and output out of the NoC infinitely often*" as:

  $\boldsymbol{G}\ \boldsymbol{F}\ input \Rightarrow \boldsymbol{G}\ \boldsymbol{F}\ output$

3.2.3 Temporal Model checking problem

Given a Kripke structure, $M = (S, S_0, R, L)$, the LTL model checking problem, $M \vDash \phi$: checks if $\langle M , S_0 \rangle \vDash \phi$, for every $S_0 \in S$, initial state of the Kripke structure M.

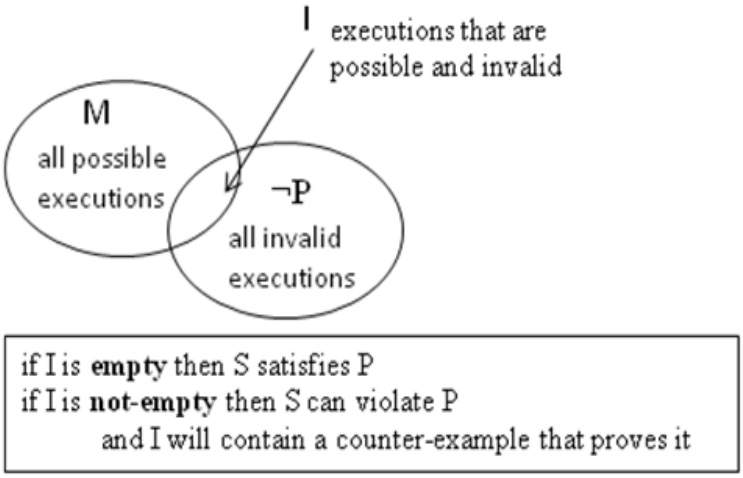
There are different techniques to perform this check. A basic method that is used by SPIN[Hol97] which is based on the automata-theoretic approach is as follows:

- For the model of the system, M, let L(M) be the set of possible behaviours of M
- For the properties, P, let L(P) be the set of valid/desirable behaviours

- We want to prove that $L(S) \subseteq L(P)$. That is everything possible is valid.
  - But language inclusion is complex [GH04].
- Therefore the method used is
  - let $L(\bar{P})$ be the language $\Sigma^{\omega} \setminus L(P)$ of words not accepted by *P*
  - Prove $L(M) \cap L(\bar{P}) = \emptyset$. That is, no accepted word by *M* is disallowed by *P*

Figure 3.2 shows a pictorial view of the model checking method described above.

Figure 0-2 Model checking problem



3.2.4 State Space Explosion

Model checking is based on constructing a graph of consisting of nodes representing states that the system can reach and edges representing all transitions that the system can make between those states. This graph is called the state space of the system. The construction of these state spaces can be done fully automatically and practical algorithms are known for anyalysing the state space. This makes model checking easy for the user to perform than theorem proving that requires a trained personnel.

Unfortunately for practical systems the state space becomes so large that explicit model checking becomes restricted by main memory size. More generally, intuitively speaking, the size of a state space of a system tends to grow exponentially in the

number of its processes and variables, where the base of the exponentiation depends on the number of local states a process has and the number of values a variable may store, and on some kind of tightness of the connection between the components of the systems (that is, the extent to which the local states of components are determined by the local states of other components)[Val98]. The problem of state space explosion is the biggest and fundamental problem of state space techniques such as model checking. Although this problem is very formidable, the advantages of state space techniques have motivated much research into ways of reducing the state space problem.

Various state space reduction techniques have been proposed and implemented, some of these are presented below.

On-the-fly: This technique integrates the algorithm that checks the validity of a property (verification) and the algorithm that constructs the state space. Since reachable state space is usually only a fraction of the whole state space and properties need only hold on reachable state space, the idea of this technique is to do a forward traversal and never visit unreachable states. The construction of the state space is therefore stopped as soon as a bad state (that is, an error against the property) is found. Since this technique tries to speed up verification by stopping when an error is found, it means that for correct systems this will not be any speedup. An incorrect system tends to have a larger state space than the corresponding correct one [Val98]. Therefore on-the-fly verification tends to reduce the state space of an erroneous system back to roughly the same level with the corresponding correct system. One way also that on-the-fly verification may reduce the state space is by avoiding the construction of irrelevant parts of the property being checked.

Abstraction: One way of also reducing the state space is by ignoring details which does not affect the property being checked. This can be done by pre-processing the model by modifying the system description before starting the construction of the

state space. Since the state space growth depends on the variable of the system, it is customary to use as few variables as possible in a verification model. It is important to note that the correctness of abstraction is the responsibility of the author of the verification model. Theories and automatic tools can also be used for pre-processing the model in such a way that the result of the verification is not affected by the abstraction used.

Symbolic model checking [McM93 , BCM92]: This technique avoids explicit construction of the state space by representing sets of states by some data structures. The transition relation of the system and the state sets are modelled as boolean functions and represented using binary decision diagrams (BDD) [Bry86]. Model checking of temporal-logical properties, then, reduces to symbolic fixpoint computation that uses BDD-based image computation as a primitive. Model checkers base on BDDs are usually able to handle billions of states.

Partial–order reduction[Maz88] : In asynchronous systems consisting of a set of communicating concurrent processes(for example, network protocols and distributed algorithms), the behaviour of the system can be modelled as the set of all interleaving of the events in individual processes. If the concurrent events are independent then all interleaving are equivalent in that they lead to the same state. Partial-order reduction techniques exploit this redundancy and visit only a subset of the reachable states.

Chapter 4

Refinement Model Checking

This chapter introduces the CSP language for describing concurrent systems and the FDR tool which is a model checker for verification of CSP models. The concept of refinement which is the heart of FDR is also introduced. Finally the ProBE tool which is an animator for CSP processes is also presented.

4.1 CSP

Communicating Sequential Processes (CSP) developed by C.A.R. Hoare [Hoa04] is a notation for describing concurrent systems (i.e. ones where there is more than one process existing at a time) whose component *processes* interact with each other by *communication*. CSP can be described as "a collection of mathematical models and reasoning methods" on concurrent systems, with developed operational, denotational and algebraic semantics [Ros97].

In the view of CSP instead of various components of a system interacting by assigning them to shared state variables, processes interact by synchronising on some common events. Strictly speaking CSP is not a programming language but a mathematical notation. However there have been a number of programming languages for describing concurrent systems which have been based on CSP such as OCCAM and Ada.

4.1.1 The CSP Language

The basic syntax of CSP is described by the following grammar:

$$
\begin{aligned}
Process ::== & \ STOP \ | \\
& \ SKIP \ | \\
& \ Event \rightarrow Process \ | \\
& \ Process; Process \ | \\
& \ Process \ |[alph]|[alph]| \ Process \ | \\
& \ Process \ ||| \ Process \ | \\
& \ Process \sqcap Process \ | \\
& \ Process \ \square \ Process \ | \\
& \ Process \setminus event \ | \\
& \ f(Process) \ | \\
& \ name \ | \\
& \ \mu \ name \bullet Process
\end{aligned}
$$

Here *event* ranges over a universal set of events, Σ, *alph* ranges over subsets of Σ, *f* ranges over a set of function names, and *name* ranges over a set of process names.

Below we explain the CSP grammar presented above.

Process: A process describes the behaviour of an object in terms of the events that it can perform. Processes may exist independently or communicate with other processes. *Events* are the visible atomic actions of a process and the set of possible events a process, P, may perform is called its *alphabet,* written α(P). For example, consider a simple light switch which can be pressed up or down. The alphabet of such a process can be written as:

$\alpha(SWITCH)=\{up,down\}$.

The simplest behaviour a process can have is to do nothing. Such a process is described as *STOP*. This process never engages in any event and therefore represents deadlock. Another basic process which does nothing but terminates successfully is *SKIP*. It only performs the special event $\surd$, which represents a successful termination.

Prefixing: The simplest way of constructing a non trivial process is by *prefixing*. If *ON* is a process, and *down* is an event, then $down \rightarrow ON$ is a process which can perform the event down and then behave like *ON*. If we give this new process a name as *SWITCH*, then process *SWITCH* can be written as:

$$SWITCH = down \rightarrow ON$$

It should be noted that the expressions $P \rightarrow Q$ and $a \rightarrow b$, *where P and Q are processes,* and *a, b* are events, are not allowed. Prefixing is only used with an event and a process**.** For the case of *P* and *Q* if you want to represent a process that behaves as *P* and then behaves as *Q*, then the sequential operator that will be discussed latter should be used. In the case of *a* and *b*, if the process only performs the event *a* followed by *b* and then terminates, it should be written as

$a \rightarrow b \rightarrow STOP$ or $a \rightarrow b \rightarrow SKIP$

depending on whether we wish to model unsuccessful or successful termination.

Recursion: Using *STOP* or *SKIP* and prefixing only allows us to model systems that terminate after a finite number of events. Most often we want to model systems that run forever or perform infinite number of events. In such situations we need to use a *recursive* definition. For example, to describe a *CLOCK* that ticks forever, we can write:

$$CLOCK = tick \rightarrow CLOCK$$

This means that the process *CLOCK* performs a tick event and starts again. It should be noted that in this definition there is no timing considered here, that is how long it takes for the next tick to be performed. All we are saying is a clock continues ticking forever. In the same way we can define a SWITCH as:

$$SWITCH1 = down \rightarrow up \rightarrow SWITCH1$$

In CSP a number of processes can be defined in terms of each other by mutual recursion. For example process *SWITCH2* can be defined in terms of process ON as:

$$SWITCH2 = down \rightarrow ON$$

$$ON = up \rightarrow SWITCH2$$

It can be seen that both processes *SWITCH1* and *SWITCH2* behaves the same.

In algebraic form, recursion can be defined as *the* solution to the equation

$$X = F(X)$$

To ensure uniqueness of solution of the solution X, the function *F* is guarded which means that $F(X)$ always starts with an event. Example $F(X) = cling \rightarrow X$ is guarded while $F(X) = X$, is not guarded. This equation $X=F(X)$ has at least one and at most one solution given as:

$$uX = F(X)$$

For example our process *SWITCH1* can be written algebraically as:

$$SWITCH1 = uX.(down \rightarrow up \rightarrow X)$$

Choice: Up until now we have considered only processes which only perform a single sequence of events and terminate or repeat. Most times we want to model systems which may have alternative behaviours, perhaps determined by their environment. Such processes may offer their environment a choice of events at some point.

If *P* and *Q* are processes and *x* and *y* are distinct events, then

$$x \rightarrow P \mid y \rightarrow Q$$

is the process which can either perform event *x* and then behaves like *P*, or perform event *y* and then behave like *Q*.

For example a simple ATM machine that accepts a card and offers its user the choice to make a withdrawal or check a balance can be described in CSP as.

$$Simple\_ATM = in\_card \rightarrow (withdrawal \rightarrow out\_card \rightarrow Simple\_ATM$$

$$\mid balance \rightarrow out\_card \rightarrow Simple\_ATM)$$

The definition of choice can readily be extended to more than two alternatives, e.g.,

$$(x \rightarrow P \mid y \rightarrow Q \mid \ldots \mid z \rightarrow R)$$

Since we will want to avoid a situation such as

$$x \rightarrow P \mid x \rightarrow Q$$

it will be wrong to write $P \mid Q$. Therefore to ensure this situation does not happen, events *x* , *y*, ..., and *z* should be distinct prefixes.

In general, if *A* is any set of events, then

$$(x : A \rightarrow P (x))$$

defines a process which first offers a choice of any event *x* in *A*, and then behaves like *P(x)*.

Composition: A system will normally consist of many processes composed together to form a single process. CSP offers different ways of composing such processes together.

One of the simple forms of composition is the sequential composition. If *P*, *Q*, *R* are processes such that *P* is the sequential composition of *Q* and *R* we write

$$P = Q \,;\, R$$

Then process *P* is a process with first behaves like *Q* but when *P* terminates successful, *P* continues by behaving as *R*. If *Q* never terminates successfully, then P will also never terminate.

Composing processes sequentially is easy as compared to composing processes that interact in parallel. This is because in parallel composition the processes will have to simultaneously perform some events; an event thus becomes a joint action in which the processes involved may participate together. It is therefore important to specify the events that the processes are supposed to be interacting on. This is where defining the *alphabets* of the processes are very necessary.

If the processes *P* and *Q* have alphabets from the sets *A* and *B* respectively, then the process

$$P_{\mathrm{A}} \|_{\mathrm{B}} Q$$

is a parallel composition of *P* and *Q*. In this parallel composition, *P* can only perform events in *A*, *Q* can also perform events in *B*, and any events in the intersection of *A* and *B* require synchronisation between *P* and *Q*.

As an example of a parallel composition, let us consider our ATM machine and a customer who wants to perform only withdrawals.

$$ATM = in\_card \rightarrow (\, withdrawal \rightarrow \; out\_card \rightarrow ATM$$

$$\mid balance \rightarrow \; out\_card \rightarrow ATM)$$

$$CUST = in\_card \rightarrow withdrawal \rightarrow CUST.$$

$$\alpha(ATM) = \alpha(CUST) = \{in\_card,\, withdrawal, balance, out\_card\} = A$$

$$\alpha(CUST) = \{in\_card, withdrawal, balance\} = B$$

Their parallel composition can be given as

$$SYSTEM = ATM_A||_B CUST$$

The behaviour of process *SYSTEM* can be described by the following equations.

$$\begin{aligned}
SYSTEM &= ATM_A||_B CUST \\
&= in_card \rightarrow (withdrawal \rightarrow out_card \rightarrow ATM \\
&\qquad | \; balance \rightarrow out_card \rightarrow ATM)\ _A||_B \\
&\quad in_card \rightarrow withdrawal \rightarrow CUST. \\
&= in_card \rightarrow ((withdrawal \rightarrow out_card \rightarrow ATM \\
&\qquad | \; balance \rightarrow out_Card_ATM)\ _A||_B \\
&\qquad withdrawal \rightarrow CUST\ ) \\
&= in_card \rightarrow withdrawal \rightarrow (out_card \rightarrow ATM\ _A||_B\ CUST) \\
&= in_card \rightarrow withdrawal \rightarrow out_card \rightarrow (ATM\ _A||_B\ CUST) \\
&= in_card \rightarrow withdrawal \rightarrow out_card \rightarrow SYSTEM
\end{aligned}$$

Both parallel operators can be extended to any number of processes. Given a sequence of processes $V = \langle P_1, P_2, ..., P_n \rangle$ with corresponding alphabets $\langle A_1, A_2, .., A_n \rangle$ their parallel composition can be written as

$$PAR(V) = ||_{i=1}^{n} (P_i, X_i) = P_1 \quad _{X_1}||_{X_2 U \ldots U X_n} (\ldots (P_{n-1} \quad _{X_{n-1}}||_{X_n} P_n) \ldots)$$

It may happen that although two or more processes run in parallel they do not synchronize on any events. In such as situation the interleaving operator is used to compose the processes together. The processes may have different common events but they do not interact. Therefore each event of the composed process is

performed by exactly one of the processes. If an event belongs to only one process then that process performs it. But in the case that the event belongs to both processes then a non-deterministic choice is made. The interleaving of processes *P* and *Q* can be written as

$$P \,|||\, Q$$

Previously when we discussed the choice operator, we did not specify whether the choice was made by the process itself or by its environment. We now distinguish between these two choices.

Internal choice: If P and Q are processes then

$$P \sqcap Q$$

denotes the process which is either *P* or *Q*; the choice is arbitrary but beyond the influence of the environment. Such a choice is referred to as *internal choice* because it is the system that makes the choice without the control of the environment. This is used to describe *non-deterministic* behaviour. The internal choice operator, $\sqcap$, allows us to describe a concurrent system in a very abstract way. The appearance of internal choice in a design allows the implementer a choice in finding an implementation. Process P $\sqcap$ Q *may be implemented by* either P or Q (a choice for the implementer).

The "*may be implemented by*" relation is called refinement which we will talk about more later in this chapter. *Refinement is thus the removal of nondeterminism*.

External choice: On the other hand an external choice operator, $\Box$, enables the behaviour of a process to be controlled by other processes running in parallel with it, collectively called its *environment* .If *P* and *Q* are processes then

$$P \Box Q$$

denotes their external choice whose initial event is determined by the environment. If it is an event of just *P* then $P \square Q$ behaves like *P*. If it is an event of just *Q* then it behaves like *Q*. If it is an event common to both *P* and *Q* then its behaviour is the nondeterministic choice between *P* and *Q*.

Both the internal and external choice operators can be written in an indexed form.

$$\square_{x:A}\, x \rightarrow P(x)$$

describes a process which offers its environment any event in the set *A* and then behaves as *Px*. While the process

$$\prod_{x:A} x \rightarrow P(x)$$

offers its environment exactly one event *x* from the set *A*; the choice being non-deterministic.

Hiding: Sometimes in describing systems we may only want to observe a relevant subset of all possible events that the process can perform. This can be done is CSP by using the *hiding* or *concealment* operator. If C is a finite set of events to be concealed, then

$$P \setminus C$$

is a process which behaves like *P* , except that each occurrence of any event in *C* is concealed or hidden.

For example consider a noisy vending machine

$$NVM = coin \rightarrow cling \rightarrow toffee \rightarrow clung \rightarrow NVM.$$

This machine can be put in a soundproof box as,

$$NVM \setminus \{cling, clung\}$$

this will become like a normal vending machine

$$VM = coin \rightarrow toffee \rightarrow VM$$

A very important thing to consider here is that hiding may introduce nondeterminism into a deterministic process. It may also make a process *diverge*; that is a situation whereby the process performs infinite sequence of hidden actions. For instance consider our process

$$CLOCK = tick \rightarrow CLOCK$$

if we hide the event tick

$$CLOCK \setminus tick$$

this process will diverge.

Communication: So far we have considered all events in the same way regardless of whether they are thought of as input or output. In CSP, an event can be described as input or output by declaring *channels* which have type, that describes the set of possible values which that channel can be transmitted along it. Events of the form *c?x* or *c!e* are called input and output events respectively. *c* is the channel, *e* the expression being output along the channel and *x* is the variable assigned a value along the input channel. The process

$$COPY = in?x{:}T \rightarrow out!x \rightarrow COPY$$

is a process which is ready to accept any value *x* of type *T*, on its input channel *in* and then output this value on the output channel *out* and then behaves as *COPY*.

Conditional: A very useful operator for describing systems is the conditional operational operator. Let b be a boolean expression (evaluates to true or false). Then

$$\mathrm{P} \lhd \mathrm{b} \rhd \mathrm{Q}$$

pronounced *P if b else Q*, is a process which behaves like *P* if *b* is true or like *Q* if *b* is false.

Parameter: Processes can be defined with parameters. Such parameters are data structures; with type and operations. Examples are integers, real numbers, events, sets, etc. For example a counter can be defined as

$$Count(n) = in \rightarrow Count(n+1)$$

$$\Box\ out \rightarrow Count(n-1) \triangleleft\ n>0 \triangleright \text{STOP}$$

We will like to conclude this section by noting that we have only looked at just some of the basic operators in CSP. We also did not talk about the laws governing these operators and their proofs. A comprehensive study of the core CSP and an extension of CSP can be found in [Hoa04 , Ros97]

4.1.2 Observing Process behaviour – Failure, Divergence and Refinement

In the previous section we just looked at the syntax of CSP without giving any formal meaning to the syntax. To be able to give a precise meaning to our operators, we should be able to describe process behaviour by some *observable properties*. CSP allows a process to be defined in terms of three important observable properties namely *trace*, *failure* and *divergence* models.

4.1.2.1 Trace model

The traces of a process *P* written as

traces (*P*) ,

denotes the set of all finite sequences of events *P* can perform, in the order in which it can perform them.

For our previous process

$$SWITCH1 = up \rightarrow down \rightarrow SWITCH1$$

$$\{<>, <up>, <up, down, up>\} \subset trace(SWITCH1)$$

Note that the empty trace, <>, is a trace of every event; corresponding to the observation when no event has been performed.

Below is a subset of useful operations on traces

- Catenation: $s \frown t$ joining trace *s* and *t* together

  Example: $\langle s_1, s_2, \ldots s_n \rangle \frown \langle t_1, t_2, \ldots, t_n \rangle = \langle s_1, s_2, s_n \ldots t_1, t_2, \ldots t_n \rangle$

- Restriction: $t \upharpoonright B$, trace *t* restricted to elements of set *B*

  Example: $\langle b, c, d, a \rangle \upharpoonright \{a, c\} = \langle c, a \rangle$

- Prefix: $s \leq t$, *s* is a prefix of *t*

  Example: $\langle a,b \rangle \leq \langle a,b,c \rangle$

- Replication: S^n, trace *s* repeated *n* times.

  Example: $\langle a,b \rangle^2 = \langle a,b,a,b \rangle$

- Length: $|s|$, length of trace *s*

  Example: $|\langle a,b,c,d \rangle| = 4$

- Count: $s \downarrow b$, count of *s* in *b*

  Example: $\langle a, c, d, c \rangle \downarrow c = 2$

Trace Refinement: In Chapter 3, we noted that one of the approaches to model checking is to check if an implementation conforms to its specification. We also stated that one of the notions of conformance is refinement; where we want to find out if the implementation is a refinement of the system specification. CSP uses this concept of refinement.

Definition of refinement relation $\sqsubseteq_T$ on processes:

$$P \sqsubseteq_T Q \text{ if and only if } traces(P) \supseteq traces(Q)$$

This is pronounced as “P is refined by Q”. In FDR this is written as [T= . The Subscript *T* indicates that refinement is with respect to traces; CSP has other forms of refinement which we will discuss soon.

P is refined by *Q*, if *Q* exhibits at most the behaviour exhibited by *P*. A very important thing to note is that for any process *P*

$$P \sqsubseteq_{\mathbf{T}} STOP$$

The main use of specification is in refinement. The specification

$$SPEC \sqsubseteq_{\mathbf{T}} P$$

limits what *P* can do, but does not require it to do anything. For example,

$$\text{SPEC} \sqsubseteq_{\mathbf{T}} \text{STOP}.$$

Specifications which simply restrict the behaviour without requiring any particular behaviour are known as safety specifications [Sch00]. They specify that nothing bad can happen, without specifying that anything good must happen. (Recall in Chapter 3 when we talked about temporal safety specifications, trace refinement is the conformance model checking equivalent). *STOP* satisfies any safety specification, i.e. *doing nothing is always safe*. Therefore trace refinement can only be used to specify safety specifications. To specify liveness properties a finer semantic model is needed, which will be our next topic of discuss.

4.1.2.2 Failure model

Given a process

$$P = a \rightarrow P \quad \text{and} \quad Q = b \rightarrow Q$$

the definition of traces will make

$$traces(P \,\square\, Q) = traces(P \sqcap Q)$$

although the two process are not the same. This is because $P \,\square\, Q$ offers its environment a choice between either perform *a* or *b* which cannot be refused. Whatever choice is made by the environment will be performed. On the other hand the decision as to whether *a* or *b* is performed in $P \sqcap Q$ is made internally without

the environment. The process may refuse either *a* or *b* but not both; whichever of them is offered by the environment, which may lead to a deadlock. Therefore a finer semantic model is needed to distinguish such processes. One of such model is the *failures model*. The failure model allows us to reason about the circumstances under which as system may deadlock. The failures of a process *P* consist of the pairs *(s, X)*, where s is a set of trace of *P* and *X* is a set of events which if offered to *P* by its environment after performing trace s, might be refused. The failures of a process describe the circumstances under which it might deadlock. A state of a process is deadlocked if it can refuse to do every event, and *STOP* is the simplest deadlocked process. Deadlock is also commonly introduced when parallel processes do not succeed in synchronising on the same event.

For example if

$$P = a \rightarrow b \rightarrow STOP$$

$$\alpha(P) = \{a, b\}$$

then

$$\text{failures(P)} = \{ (\langle\rangle,\{\}),(\langle\rangle,\{b\}), (\langle a\rangle,\{\}), (\langle a\rangle,\{a\}), (\langle a,b\rangle,\{\}),(\langle a,b\rangle,\{a\}), (\langle a,b\rangle,\{b\}), (\langle a,b\rangle,\{a,b\})\}$$

A process *P* can deadlock after trace *s* if and only if $(s, \Sigma) \in failures(P)$. That is, after performing s the network refuses to perform any event its environment offers it. *P* is deadlock-free if

$$\forall s: traces(P). \quad (s, \Sigma) \notin failures(P)$$

A network *V* is said to be deadlock-free if the process *PAR(V)* is deadlock-free.

Failures Refinement: Definition of failures refinement relation $\sqsubseteq_F$ on processes:

$$P \sqsubseteq_F Q \text{ if and only if } failures(P) \supseteq traces(Q)$$

4.1.2.3 Failure-Divergence model

A process livelocks if it performs an infinite sequence of internal actions which means it will not engage in any visible event. Therefore the failures model is not enough to reason about such processes. A new model, failures-divergence model is used to reason about such processes by adding the concept of divergences. The divergences of a process are a set of the traces after which it might diverge. Example

$$<> \in divergences(CLOCK \setminus tick)$$

The failure-divergence relation, written, $\sqsubseteq_{FD}$ is defined as:

$$P \sqsubseteq_{FD} Q = failures(Q) \subseteq \text{failures}(P) \wedge$$

$$\text{divergences}(Q) \subseteq divergences(P)$$

For a divergence free processes which most practical systems are, using both $\sqsubseteq_{FD}$ and $\sqsubseteq_F$ will produce the same results

4.1.3 Operational Semantics

We have so far looked at the algebraic semantics of CSP by introducing some operators and defining a set of algebraic laws on them from which process equivalence between CSP processes can be derived. We also gave a denotational semantics of CSP by mapping processes into some abstract models namely, trace, failure and failure-divergence models. What we have not shown yet is how our CSP processes can be implemented on a machine. For that we need an operational semantics which defines how our processes should be executed. The operational semantics of CSP is a mapping from CSP expressions to labelled transitions systems (LTS). (We have already giving a formal definition of LTS in Chapter 3).

LTS is used to precisely describe processes by defining transition rules for all CSP operators. Transition or inference rules are of the form

$$\frac{hypothesis_1, \ldots, hypothesis_n}{conclusion} \; [sid\_condition]$$

Here we will look at the transition rules for some basic operators. Schneider [Sch00] provides a complete list of all transition rules. Note some rules may have no hypothesis and/or side conditions

- Prefix

$$\frac{}{(a \rightarrow P) \overset{a}{\rightarrow} P}$$

- Menu choice

$$\frac{}{x{:}A \rightarrow P(x) \overset{a}{\rightarrow} P(a)} \; [a \in A]$$

- Internal choice

$$\frac{}{(P \sqcap Q) \overset{\tau}{\rightarrow} P}$$

$$\frac{}{(P \sqcap Q) \overset{\tau}{\rightarrow} Q}$$

- External choice

$$\frac{P \overset{a}{\rightarrow} P'}{(P \,\Box\, Q) \overset{a}{\rightarrow} P'} \; [a \neq \tau]$$

$$\frac{Q \xrightarrow{a} Q'}{(P \,\square\, Q) \xrightarrow{a} Q'}\ [a \neq \tau]$$

$$\frac{P \xrightarrow{\tau} P'}{(P \,\square\, Q) \xrightarrow{\tau} (P \,\square\, Q')}$$

$$\frac{Q \xrightarrow{\tau} Q'}{(P \,\square\, Q) \xrightarrow{\tau} (P' \,\square\, Q)}$$

- Parallel composition

$$\frac{P \xrightarrow{a} P'}{(\ P\ |[A|B]|Q\ ) \xrightarrow{a} (\ P'|[A|B]|Q\ )}\ \left[a \in (A - B - \{\surd\}\ ) \cup \{\tau\}\right]$$

$$\frac{Q \xrightarrow{a} Q'}{(\ P\ |[A|B]|Q\ ) \xrightarrow{a} (\ P'|[A|B]|Q\ )}\ \left[a \in (B - A - \{\surd\}\ ) \cup \{\tau\}\right]$$

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{(\ P\ |[A|B]|Q\ ) \xrightarrow{a} (\ P'|[A|B]|Q\ )}\ \left[a \in (A \cap B\ ) \cup \{\surd\}\right]$$

- Sequential composition

$$\frac{P \xrightarrow{a} P'}{(P\ ;\ Q) \xrightarrow{a} (P';\ Q)}[a \neq \surd]$$

$$\frac{P \xrightarrow{\surd} P'}{(P\ ;\ Q) \xrightarrow{a} Q}[a \neq \surd]$$

- Interleaving

$$\frac{P \xrightarrow{a} P'}{P\ |||\ Q \xrightarrow{a} P'\ |||\ Q}[a \neq \surd]$$

$$\frac{Q \xrightarrow{a} Q'}{P \,|||\, Q \xrightarrow{a} P \,|||\, Q'}\,[a \neq \surd]$$

$$\frac{P \xrightarrow{\surd} P' \quad Q \xrightarrow{\surd} Q'}{P \,|||\, Q \xrightarrow{\surd} P' \,|||\, Q'}$$

- Hiding

$$\frac{P \xrightarrow{a} P'}{(P \setminus A) \xrightarrow{\tau} (P' \setminus A)}\,[a \in (A \cup \{\tau\})]$$

$$\frac{P \xrightarrow{a} P'}{(P \setminus A) \xrightarrow{a} (P' \setminus A)}\,[a \notin (A \cup \{\tau\})]$$

- Recursion

$$\frac{}{\mu X \bullet F(X) \xrightarrow{a} F(\mu X \bullet F(X))}$$

4.2 FDR

FDR(Failure-Divergence Refinement) is the de facto model checker for the CSP language from Formal Systems Europe Limited [FSEL]. Its method of establishing whether a property holds is to test for the refinement of a transition system capturing the property by the candidate machine.

The main ideas behind FDR are presented in [R94]. FDR currently supports all the three refinement models, trace, failures, failure-divergence, described in the previous section.

FDR can be used to for checking refinement, deadlock, livelock and determinism. The new version FDR2 has been designed to improve on the flexibility and scalability of the tool [FDR05]. The input language to FDR is the machine-readable dialect of CSP, called CSP_M. CSP_M combines the CSP process-algebra with an expression language which has been adapted to support the idioms of CSP. The syntax of CSP_M together with some illustrative examples can be found in the FDR2 manual[FDR05].

Below is an example of a CSP_M file which can be input into FDR2 for model checking.

```
num_servers = 2
channel enter,serve : {1..num_servers}
channel join
N=3

QUEUE =
 let
        Q(0) = join -> Q(1)
        Q(n) = join -> ( if (n<N) then Q(n+1) else Q(n) )
                []
                ( if (n>0) then enter -> Q(n-1) else STOP )
 within Q(0)

SERVER = enter -> serve -> SERVER

SYSTEM = QUEUE [{join,enter}||{enter,serve}] SERVER
```

This is a CSP model of a system consisting of a queue and a server. There are two servers and the queue size is 3. When the queue is full no entry is allowed. The *SYSTEM* is a parallel composition of the *QUEUE* and *SERVER* processes synchronizing on the *enter* event.

4.2.1 Verification with FDR

The FDR tool accepts a CSP script file which is a file containing CSP*M* code with the file extension "*.csp*" or "*.fdr*". There are two different ways of using FDR to check assertions. The first one method is to include the assertion statements in the FDR

CSP script. When such a script is loaded, all assertions will be displayed in the FDR tool window. Assertions normally have the following form:

assert SPEC [X= IMPL

where *SPEC* is the process representing the specification of the system and *IMPL* is a process representing the model of the system to be verified. The *X* indicates the type of refinement model to be used: *T* for traces, *F* for failures and *FD* for failures-divergence. If *X* is not specified FDR assumes failures-divergence (*FD*) which is the default refinement model.

The second method for specifying assertions to be checked by FDR is by using the Specification and Implementation dropdown on the main window of FDR tool as shown in Figure 4.1 to select the processes.

A very useful feature of the FDR tool is that deadlock, livelock and determinism can easily be checked by selecting the appropriate tab and then selecting the process to be checked from the implementation dropdown or the process list area. See Figure 4.1.

Once the specification (in the case of refinement) and implementation process have been selected, the checking is done by simply clicking the 'check' button. Depending on the system being verified the results of the verification can be in seconds, minutes or even hours.

When the checking finishes or is interrupted, a symbol will be placed beside the assertion checked (Figure 4.1).

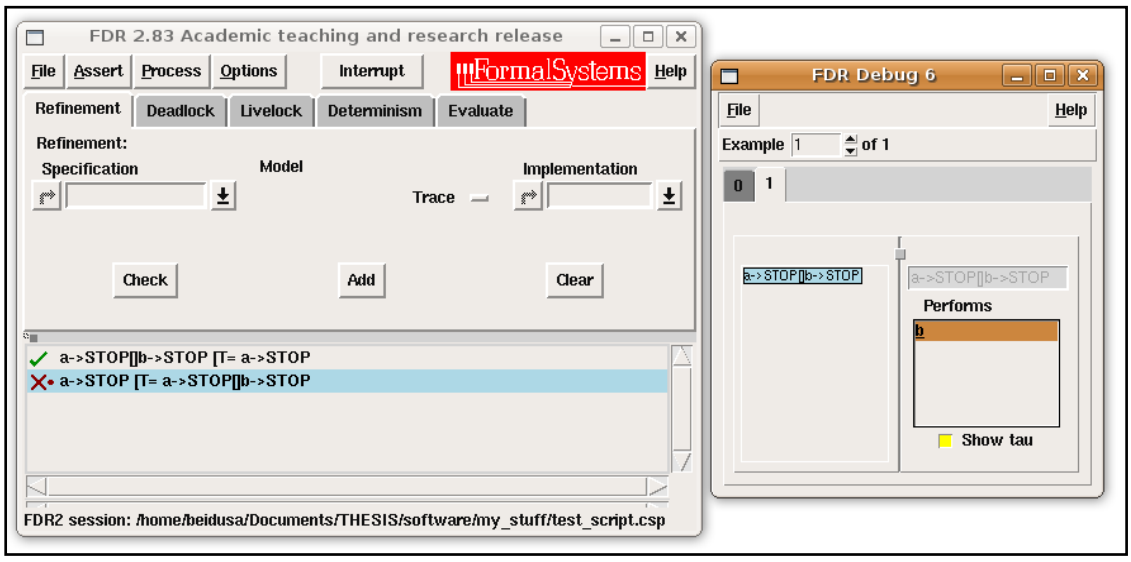
- **Green tick:** indicates that the check completed successfully; the stated refinement holds.
- **Red cross**: indicates that the check completed, but found one or more counterexamples to the stated property: the refinement does not hold, and the FDR debugger can be used to explore the reasons why.

- **Red Exclamation**: indicates that the check failed to complete for some reason; either a syntax or type error was detected in the scripts, some resource was exhausted while trying to run the check, or the check was interrupted. If a process could not be compiled, FDR will also indicate this by popping up a warning dialogue box.
- **Zig Zag**: indicates that FDR was unable to complete a check because of a weakness in the currently coded algorithms. (This can occur under rare circumstances when performing a determinism check in the Failures model. I have never experienced this situation in my usage of FDR2)

We have already mentioned that one of the main advantages of model checking over other techniques is the fact that a counterexample is produced for checks that fail. FDR provides a debugging window that helps to understand why a check failed by providing counterexamples. A description and an example of how to use the debugging windows for counterexamples can be found in the FDR2 manual [FDR05].

As a simple example of using FDR debugging window to view a counterexample, consider Figure 4.1 showing the results of checking two assertions shown.

Figure 0-1 Counter-example in FDR



Let

$$traces(\ a \rightarrow STOP \ \square \ \ b \rightarrow STOP) = \{<>,<a>,<b>\} = P$$

$$trace(a \rightarrow STOP) = \{<>,<a>\} = Q$$

Now coming back to the example shown in Figure 4.1, the first assertion is

$$a \rightarrow STOP \ [] \ b \rightarrow STOP \ \ [T= \ a \rightarrow STOP$$

This assertion is true because

$$Q \subseteq P$$

On the other hand the second assertion

$$a \rightarrow STOP \ \ [T= \ a \rightarrow STOP \ [] \ b \rightarrow STOP$$

is false because

$$P \not\subseteq Q$$

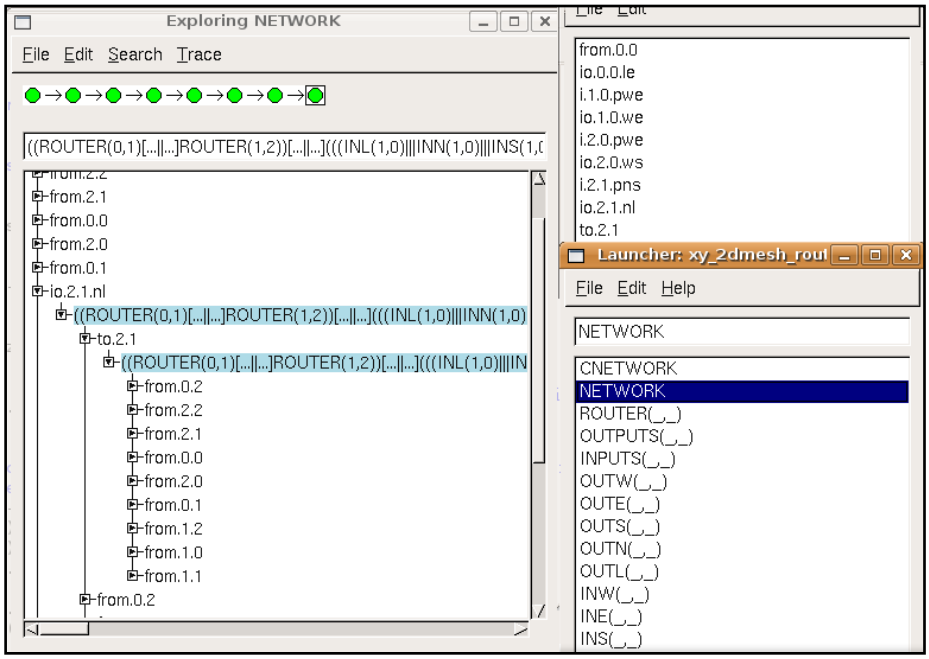
Now let us look at the debugging window of FDR shown in Figure 4.1. We notice the event *b* in the *Performs* section. What this means is that $(a \rightarrow STOP \ [] \ b \rightarrow STOP\ )$ performed the event *b* which is not in the specification $(a \rightarrow STOP)$, and therefore it is the counterexample for the second assertion that failed to verify. Obviously for complex systems the counterexample may not be easy to see like this example.

4.3 ProBe

We will like to end this chapter by introducing a very useful and handy tool for CSP authors. ProBE(Process Behaviour Explorer) is an animator for CSP processes also developed by Formal Systems Europe Limited**[FSEL]**. The ProBE tool enables the user to “browse” a CSP process by following events that lead from one state of the process to another. It uses a hierarchical list to display the possible actions and states of a process in much the same way as a file system viewer shows directories

and files. It is a very useful tool as it helps to identify design errors in CSP processes. Figure 4.2 shows a snapshot of the ProBE tool being used to animate a CSP process.

Figure 0-2 Using ProBE to animate a process



Chapter 5

Probabilistic Model Checking

5.1 Introduction

Probabilistic model checking refers to a range of techniques for calculating the likelihood of the occurrence of certain events during the execution of the system [KNP02]. These techniques are useful to establish properties such as “input buffer gets full with a probability of at most 0.1” and “there is a probability of 1 that once a flit enters a router it will come out”. As have been described previously in this report, in conventional model checkers, the model of a system is created as a state transitions system and the properties to be verified are stated in some temporal logic. In the case of probabilistic model checking, transitions between states of the model are probabilistic. That is, while in conventional models we only speak of transition between states S_1 to S_2, in probabilistic models, we speak of transition between states S_1 and S_2 with probability of 0.5. An example of a probabilistic model checker is the PRISM model checker developed by Kwiatkowska et al. [KNP02] which was used in this work.

PRISM is a model checker that supports both Discrete Time (DTMC) and Continuous Time (CTMC) Markov chains, and Markov Decision Process [Der70].

5.1.1 Probabilistic model checking

Formally, a Markov chain[Reference] can be defined as the tuple:

$$(S, S_0, T, L)$$

where

- *S* is a set of states
- $S_0 \in S$ is the initial state
- $T : S \times S \rightarrow [0,1]$ is transition relation such that
  - $\forall s, s' \in S, \ \Sigma_{s'} T(s, s') = 1$
- $L: S \rightarrow 2^{AP}$ is a labelling function from states *S* to a finite set of atomic propositions.

Discrete-time Markov chains (DTMCs) specify the probability π(s, s') of making a transition from state *s* to some target state *s'*, where the total probabilities of reaching a target state must sum up to 1, i.e. $\Sigma_{s'} \pi(s, s') = 1$.

Continuous-time Markov chains (CTMCs), on the other hand, specify the rates *ρ(s, s')* of making a transition from state *s* to *s'*, with the interpretation that the probability of moving from *s* to *s'* within $t \in \mathbb{R}^{>0}$ time units is $1 - e^{-\rho(s, s') \cdot t}$.

Markov decision processes (MDPs) extend DTMCs by allowing both probabilistic and non-deterministic behaviour. Non-determinism enables the modelling of asynchronous parallel composition, and permits the under-specification of certain aspects of a system [KNP02].

Properties to be verified on a probabilistic model are specified in Probabilistic Computation Tree Logic (PCTL)[HJ94, BK98, BA95] and Continuous Stochastic Logic (CTL)[BHH00]. PCTL is used for specifying properties of DTMCs and MDPs while CSL an extension of PCTL, used for CTMCs. The PRISM model for this work was an MDP and therefore we give a brief overview of PCTL here.

5.1.2 Overview of PCTL

PCTL is used for reasoning about probabilistic temporal properties of probabilistic finite state spaces.

The syntax of PCTL

$$\Phi ::= true \mid false \mid a \mid \neg\Phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid P_{\sim p}[\varphi]$$

$$\varphi ::= X\,\Phi \mid \Phi\, U^{\leq k}\, \Phi$$

where *a* is an atomic proposition, $\sim \in \{<, \leq, \geq, >\}$, $p \in [0, 1]$ and $k \in N \cup \{\infty\}$. PCTL formulae are interpreted over the states of a DTMC.

For the presentation of the syntax above, we distinguish between state formulae Φ and path formulae φ, which are evaluated over states and paths respectively. To specify a property of a DTMC, we always use a state formula; path formulae only occur as the parameter of the $P_{\sim p}[\cdot]$ operator. Intuitively, a state *s* of DTMC model satisfies $P_{\sim p}[\varphi]$ if the probability of taking a path from *s* satisfying φ is in the interval specified by ~p.

As path formulae we allow the *X* ('next') and $U^{\leq k}$ ('bounded until') operators which are standard in temporal logic. The unbounded until is obtained by taking k equal to ∞, i.e. $\Phi\, U\, \Psi = \Phi\, U^{\leq \infty}\, \Psi$.

Intuitively, $X\,\Phi$ is true if Φ is satisfied in the next state and $\Phi\, U^{\leq k}\, \Psi$ is true if Ψ is satisfied within k time-steps and Φ is true up until that point.

For a state *s* and PCTL formula Φ, we write $s \vDash \Phi$ to indicate that *s* satisfies Φ. Similarly, for a path ω satisfying path formula φ, we write $\omega \vDash \varphi$.

5.2 PRISM language

The behaviour of a system to be verified by PRISM model checker is specified using a simple module-based language inspired by Reactive Modules formalism of Alur and Henzinger[AH96].

The fundamental components of the PRISM language are *modules* and *variables*. A model is composed of a number of *modules* which can interact with each other. A module contains a number of local *variables*. The values of these variables at any given time constitute the state of the module. The *global state* of the whole model is determined by the *local state* of all modules. The behaviour of each module is

described by a set of *commands*. A command takes the form:

[] guard -> prob_1 : update_1 + ... + prob_n : update_n;

The *guard* is a predicate over all the variables in the model (including those belonging to other modules). Each *update* describes a transition which the module can make if the guard is true. A transition is specified by giving the new values of the variables in the module, possibly as a function of other variables. Each update is also assigned a probability (or in CTMCs a rate) which will be assigned to the corresponding transition.

State variables are declared in the standard way. For example, the following declaration

full: bool init false;

declares a boolean state variable *full*, initialized to false, while the following declaration

```
const BufferSize = 4;
...
NorthBuffer: [0..TotalRuns-1] init 0;
```

declares a constant *BufferSize* equal to 4, and then an integer array of size 4, indexed from 0 to *BufferSize*-1, with all elements initialized to 0.

Modules are specified as :

```
module name
...
endmodule
```

For example :

```
module M1
    x : [0..2] init 0;
    [] x=0 -> 0.8:(x'=0) + 0.2:(x'=1);
    [] x=1 & y!=2 -> (x'=2);
    [] x=2 -> 0.5:(x'=2) + 0.5:(x'=0);
endmodule
```

In PRISM the model of system is constructed as the parallel composition of its modules. Modules can also be composed so that they synchronise on some specific actions by using CSP-based operators. For example the following can be done

- $M1 \;||\; M2$: full parallel composition of modules *M1* and *M2* (synchronising on all actions which appear in both *M1* and *M2*)
- $M1 \;|||\; M2$: asynchronous parallel composition of *M1* and *M2* (fully interleaved, no synchronisation)
- $M1 \;|[a,b,...]|\; M2$: restricted parallel composition of modules *M1* and *M2* (synchronising only on actions from the set {*a*, *b*,...})
- $M \,/\, \{a,b,...\}$: hiding of actions {*a*, *b*, ...} in module *M*
- $M\ \{a<\text{-}b,c<\text{-}d,...\}$: renaming of actions *a* to *b*, *c* to *d*, etc. in module *M*.

These are specified using the **system ... endsystem** construct placed at the end of the model description.

Modules which have the same behaviour can be constructed easily by using module *renaming*. For example in this work when modelling the NoC routers, since all corner routers have the same behaviour only one was specified and module renaming was used to specify the rest. A new module *M2* can be specified by using the definition of *M1* as

$$module\ M2 = M1\ [\ x1=x2, y1=y2\ ]\ endmodule$$

where the local variables *x1* and *x2* in *M1* are renamed to *x2* and *y2* respectively in *M2*.

As stated earlier, PRISM supports three different types of probabilistic models and this must be indicated at the top of the model description file. PRISM uses the keywords **dtmc**, **ctmc** or **mdp** to indicate the type of model being described. Alternatively the keywords **probabilistic**, **stochastic** and **nondeterministic** can be used for **dtmc**, **ctmc** and **mdp** respectively.

Cost and Rewards: A very useful feature of the current version of PRISM is its support for the specification and analysis of properties based on costs and rewards. This allows PRISM to be used to reason about quantitative measures of system behaviour such as expected values. For instance, costs and rewards can be used to compute properties such as “expected time”, “expected number of flits in a network” and “expected number of drop packets”. This can be done by associating real values with certain states or transitions of the model. Although cost is normally perceived to be bad while reward is “good”, practically there is no difference between them. It is therefore up to the user to interpret the results the way he/she wants.

Rewards are declared using **rewards ... endrewards** construct and can appear anywhere in the model file except in the definition of a module. An example of a reward declaration is:

```
reward
      x = 0 : 1;
    [in] true:1;
Endrewards
```

which assigns a reward of 1 to states where *x=0* and a reward of 1 to all transitions labelled with action *in.*

5.3 PRISM Properties specification

As stated earlier properties of probabilistic models can be expressed using PCTL and CSL. PRISM uses three operators namely the **P**, **S** and **R** operators for specifying such properties. Not all the three operators can be applied to the three different probabilistic model types. Below we describe each of the operators and in each case state the model types that it can be used for.

5.3.1 The P operator

The P operator is one of the most important operators in the PRISM property specification language which is used to reason about the probability of an event's occurrence. This operator supports all three model types: DTMCs, MDPs and CTMCs.

The syntax for using the P operator to specify a property is:

$$P\ bound\ [pathprop]$$

which is true in a state *s* of a DTMC, MDP or CTMC if "the probability that path property *pathprop* is satisfied by the paths from state *s* meets the bound, *bound*". The bounds can be $\geq p$, $>p$, $\leq p$, or $<p$, where *p* is a PRISM language expression evaluating to a double in the range [0,1].

It is important to note that for an MDP, properties using the **P** operator reason about the *minimum* or *maximum* probability, over all possible resolutions of nondeterminism, that a certain type of behaviour is observed. This depends on the bound attached to the **P** operator: a lower bound (> or >=) relates to minimum probabilities and an upper bound (< or <=) to maximum probabilities.

The **P** operator can be used with different types of path properties. These include **X** for ne**x**t, **U** for until, **F** for eventually (or **F**uture), **G** for always (or **G**lobally), **W** for **W**eak until and **R** for **R**elease. An example of **P** operator property specification is:

$$P>=0.99\ [\ G\ z<10\ ]$$

which is true in a state “if there is a probability of at least 0.99 that z will never exceed 10”.

With the exception of the **X**, a time bound can be imposed on the property being specified. For example:

P>=0.98 [F<=7 y=4]

is true in a state if "the probability of y being equal to 4 within 7 time steps is greater than or equal to 0.98"

Another important feature of PRISM is its support for probabilistic model checking of LTL properties. An example is

P>=1 [G F *"input"*]

which states that "a flit is input into the network infinitely often with probability 1"

5.3.2 The S operator

The **S** operator is used to reason about the *steady-state* behaviour of a model, i.e. its behaviour in the *long-run* or *equilibrium*. Although the **S** operator can be applied to all three model types, at the time of writing this report, PRISM only supports DTMC and CTMC.

The property:

S *bound* [*prop*]

is true in a state *s* of a DTMC or CTMC if "starting from *s*, the steady-state (long-run) probability of being in a state which satisfies the (boolean-valued) PRISM property *prop*, meets the bound, *bound*". An example is:

S<0.05 [*queue_size / max_size* > 0.75]

which means: "the long-run probability of the queue being more than 75% full is less than 0.05".

5.3.3 Rewards-based properties - R operator

Earlier we looked at how to include reward information in the model description. PRISM can analyze properties which relate to the *expected values* of these rewards.

This is done using the **R** operator which works in the same way as the **P** and **S** operators. This can be specified as

R bound [*rewardprop*]

which is true in a state of a model if " the expected reward associated with *rewardprop* of the model when starting from that state meets the bound *bound*".

There are four different types of reward properties, namely:

- "**reachability** reward": **F** prop ,refer to the reward accumulated along a path until a certain point is reached.
- "**cumulative** reward" : **C**<=t , associate a reward with each path of a model, but only up to a given time bound.
- "**instantaneous** reward" : **I**=t ,refer to the reward of a model at a particular instant in time.
- "**steady-state** reward" : **S ,**relate not to paths, but rather to the reward in the long-run.

5.3.4 Quantitative properties

So far we have only being looking at properties which can be verified to be *true* or *false* in a model. But often it is useful to compute the actual probability value that some behaviour of a model is observed. This can be achieved by replacing *bound* in the properties we have seen so far with *=?* for DTMC and CTMCs and *min=?* or *max=?* for MDPs. For example

P=? [F x=5 & y=5]

returns the probability of, from the initial state, reaching a state satisfying *x=5&y=5.*

Another example is

R=? [C<=15.5]

which would return, for a given state of the model, "the expected number of dropped packets within 15.5 time units of operation".

An example for an MDP is

$$P_{max=?}\ [\ F{<=}T\ dropped\_packets > 10\ ]$$

which returns "the maximum probability that more than 10 flits have been dropped by time T" .

Chapter 6

Model Checking OASIS with FDR

To be able to verify if our OASIS NoC is free from deadlock we first needed to create a model of the system as a CSP process and then input the CSP script into FDR for model checking. Since FDR has an inbuilt deadlock checker, no specification process for the deadlock-free property needs to be specified.

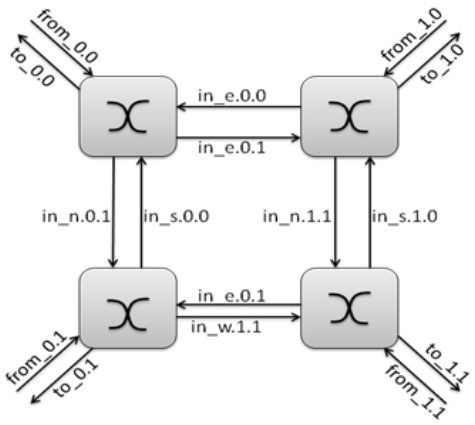
Here instead of defining our processes in the CSP algebraic construct, the definitions of our processes are given in the ASCII based syntax used by FDR.

6.1 Modeling OASIS in CSP

The CSP model of the OASIS NoC is very similar to the actual design of the OASIS NoC whereby the network is made up of an interconnected mesh of routers. In our CSP model each router was modeled as a separate process and then they were composed together in parallel to form the network process. Every router process is made up of input port processes. For the reason of time and space a very small network consisting of four flit routers connected in 2×2 mesh topology was used as for the case study. Figure 0-1 shows the design of the OASIS NoC CSP model.

Because of the geometrical nature of NoCs, one would expect to define the geometrical variations of processes by using process renaming. But this approach was used in defining our CSP processes because defining the geometry-sensitive renaming functions is no simple than just giving the variations themselves in our case, as this would have involved using a lot of conditional statements.

FIGURE 0-1 OASIS NoC CSP model digram



6.1.1 Model Parameters and channels

The only parameters of our model are the size of the network and the size of the input buffers. Because the network is 2D-mesh the size of the network is specified by the number of rows and columns of routers. To have a smaller state space that could be verified easily we chose a 2×2 network with a FIFO buffer size of just 1 flit.

```
xsize = 2 ;
ysize = 2 ;
fsize = 1 ;
```

In CSP scripts, events that a process can perform are declared using channels and the type of channels must be specified. The main events for our model are those representing movement of flits from one router to another. There are four indexes of these events; the first two representing the coordinates of the receiving input router and the last two representing the coordinates of the destination router of the flit. These are declared as:

```
Xvals : {0..xsize-1}
Yvals : {0..ysize-1}
channel from,in_n,in_s,in_e,in_w,to : Xvals.Yvals.Xval.Yvals
```

The *from* and *to* channels represent the events of transmitting a flit to and from the network respectively.

6.1.2 Input Buffer model

We next define a process for each of the five input buffers of each router of the network. Each input port consists of a *buffer* to store flits and a *route* component to determine the output direction of flits. We define separate processes for each of these two components. The input buffer is basically a finite sized FIFO(First In First Out) queue whose size is specified by *fsize*. Each input buffer can either receive a flit or send a flit to the *route* process to determine where to send the flit.

Since in reality blocks do not send flits to themselves, in the definition of the local input ports, the address of the router is excluded from the set of possible destinations of a flit.

```
INL(x,y,s) =
   letFIFO(<>)= from.x.y?xd.yd:{x'.y'|x'<-Xvals,y'<-Yvals, not(x==x'andy==y')} -> FIFO(<(xd,yd)>)
   FIFO(<(a,b)>^s)= (#(<(a,b)>^s)< fsize) & from.x.y?xd.yd:{x'.y'|x'<-Xvals,y'<-Yvals,not(x==x' and
                    y==y')} -> FIFO(<(a,b)>^s^<(xd,yd)>)  [] (#(<(a,b)>^s) > 0) & ( RL(x,y,a,b,s))
 within FIFO(s)
```

The N, S, E and W input buffers have similar definitions. As we have already noted, not all routers have all five input ports and therefore the presence of an input port in a router is dependent on the position of the router on the network. For instance for our 2×2 network example, the top row routers will not have north input ports the same way as the first column of routers will not have west input ports. This restriction is specified in the last line for each of the definitions.

```
INN(x,y,s) =
        let FIFO(<>) = in_n.x.y.x?yd:{y'|y'<-{0..(ysize-1)},y' >=y } -> FIFO(<(x,yd)>)
        FIFO(<(a,b)>^s) = (#(<(a,b)>^s) <fsize)& in_n.x.y.x?yd:{y'|y'<-{0..(ysize-1)},y'>=y}
        -> FIFO(<(a,b)>^s^<(x,yd)>) [] (#(<(a,b)>^s) > 0) & (RN(x,y,a,b,s))
 within y>0 & FIFO(s)
```

```
INS(x,y,s) =
  let FIFO(<>) = in_s.x.y.x?yd:{y'|y'<-{0..(ysize-1)},y'<=y } -> FIFO(<(x,yd)>)
  FIFO(<(a,b)>^s) = (#(<(a,b)>^s) < fsize) & in_s.x.y.x?yd:{y'|y'<-{0..(ysize-1)},y'<=y } ->
                        FIFO(<(a,b)>^s^<(x,yd)>) [] (#(<(a,b)>^s) > 0) & ( RS(x,y,a,b,s))
 within y<(ysize-1) & FIFO(s)


INE(x,y,s) =
  let FIFO(<>) = in_e.x.y?xd:{x'|x'<-{0..(xsize-1)},x'<=x }?yd -> FIFO(<(x,yd)>)
  FIFO(<(a,b)>^s) = (#(<(a,b)>^s) < fsize) & in_e.x.y?xd:{x'|x'<-{0..(xsize-1)},x'<=x }?yd ->
                        FIFO(<(a,b)>^s^<(x,yd)>) [] (#(<(a,b)>^s) > 0) & ( RE(x,y,a,b,s))
 within x<(xsize-1) & FIFO(s)


INW(x,y,s) =
  let FIFO(<>) = in_w.x.y?xd:{x'|x'<-{0..(xsize-1)},x'>=x }?yd -> FIFO(<(x,yd)>)
  FIFO(<(a,b)>^s) = (#(<(a,b)>^s) < fsize)& in_w.x.y?xd:{x'|x'<-{0..(xsize-1)},x'>=x }?yd ->
        FIFO(<(a,b)>^s^<(x,yd)>) [] (#(<(a,b)>^s) > 0) & (RW(x,y,a,b,s))
 within x>0 & FIFO(s)
```

6.1.3 Route model

After a flit has entered a router and it has been buffered, the next action is to determine which router to route the flit to; this is done in the *route* process. The route process implements the XY routing algorithm of the OASIS NoC. Our CSP *route* process does not do lookahead routing as is done in the actual OASIS NoC verilog implementation; the decision of the next route is taking in the current router. The XY routing with X-first algorithm used to determine the next port is as given below:

```
if (Xaddr = Xdest)
        if (Yaddr = Ydest)
                output = SELF
        else if (Yaddr < Ydest)
                output = NORTH
```

```
        else
                output = SOUTH
else if (Xaddr < Xdest)
        output = EAST
else
        output = WEST
```

The *local route* process can send packets to any port apart from the local output port. Its definition is given below:

```
RL(x,y,xd,yd,s) = if (xd<x) then in_e.(x-1).y!xd!yd -> INL(x,y,s)
                  else ( if (xd>x) then in_w.(x+1).y!xd!yd -> INL(x,y,s)
                         else ( if(yd<y) then in_s.x.(y-1)!xd!yd -> INL(x,y,s)
                                else ( if(yd>y) then in_n.x.(y+1)!xd!yd -> INL(x,y,s)
                                       else STOP ) ) )
```

For the other *route* processes the XY routing with X first algorithm does not permit certain movements. This means that flits have to move horizontally till they reach their X destination address before moving vertically in the network. Because of this restriction the *North* and *South route* processes can only route vertically upward or downward respectively, or to the local output port.

```
RN(x,y,xd,yd,s)  =  if (yd>y) then in_n.x.(y+1)!xd!yd -> INN(x,y,s)
                    else to.x.y!xd!yd -> INN(x,y,s)
```

```
RS(x,y,xd,yd,s)  =  if (yd<y) then in_s.x.(y-1)!xd!yd -> INS(x,y,s)
                    else to.x.y!xd!yd -> INS(x,y,s)
```

On the other hand the East and West ports can route flits in any direction except the direction from which the flit is coming from. The definition of the *East* and *West route* processes are as follows:

```
RE(x,y,xd,yd,s) = if (xd<x) then in_e.(x-1).y!xd!yd -> INE(x,y,s)
                  else ( if(yd<y) then in_s.x.(y-1)!xd!yd -> INE(x,y,s)
                         else ( if(yd>y) then in_n.x.(y+1)!xd!yd -> INE(x,y,s)
                                else to.x.y!xd!yd -> INE(x,y,s)  )  )

RW(x,y,xd,yd,s) = if (xd>x) then in_w.(x+1).y!xd!yd -> INW(x,y,s)
                  else ( if(yd<y) then in_s.x.(y-1)!xd!yd -> INW(x,y,s)
                         else ( if(yd>y) then in_n.x.(y+1)!xd!yd -> INW(x,y,s)
                                else to.x.y!xd!yd -> INW(x,y,s)  )
```

As can be seen from all the *route* processes, a flit only gets removed from the router's input buffer only after it has been moved to the next router in its path.

6.1.4 Router model

Our model of a router is simply the interleaving of all the input port processes with the initial buffer of each input port initialized to empty. This is because the input processes are independent of each other and do not synchronize on any events.

```
ROUTER(x,y) = INL(x,y,<>) ||| INN(x,y,<>) ||| INS(x,y,<>) ||| INE(x,y,<>) ||| INW(x,y,<>)
```

Note that the route processes are included in the definition of the input buffer processes.

6.1.5 Network model

Finally to obtain our network process we need to put all the router processes in parallel allowing them to synchronize on common events. To do so we first need to define an expression for the alphabet of each of the router processes.

```
Alpha(x,y) =  {| from.x.y, to.x.y, in_n.x.y, in_s.x.y, in_e.x.y, in_w.x.y,
        in_n.x.((y+1)%ysize), in_s.x.((y-1)%ysize), in_e.((x-1)%xsize).y,
        in_w.((x+1)%xsize).y|}
```

Our network process is now composed as

NETWORK = || x:Xvals, y:Yvals @ [Alpha(x,y)] ROUTER (x,y)

6.2 Verification in FDR

We are interested in verifying whether our CSP model of the OASIS NoC is free from deadlock. Verifying a process for deadlock in FDR is as simple as stating a deadlock assert statement or using the FDR2 gui tool. We can write an assert statement for deadlock check of our network process as

assert NETWORK : [deadlock free [F]]

Once the assert statement above has been inserted into the CSP script, loading the script into FDR will display the deadlock specification in the FDR tool window which can then be double clicked to perform the check.

It is also possible to still check deadlock freedom of a model in FDR without writing an assert statement by using the deadlock check window in FDR.

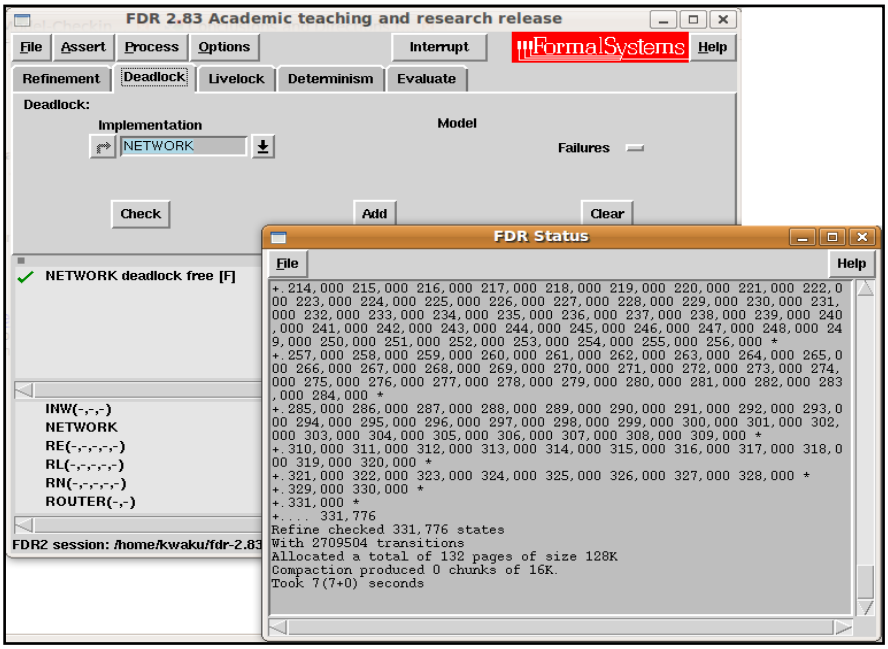
6.3 FDR Verification Results and Analysis

Verifying the CSP model described above for deadlock-freedom using the FDR tool showed that the OASIS NoC model was deadlock free as can be seen in **Figure 0-2**. From **Figure 0-2** the FDR status window shows that FDR checked 331, 776 states and the whole verification took just 7 seconds on an Intel Core 2 Duo 1.80GHz CPU with 2038MB of RAM. It should be recalled that this network is only a 2×2 mesh.

Adding just one more row and column to the network to obtain a 3×3 mesh network showed really interesing results. When the 3×3 mesh CSP model was loaded into FDR it took over 10 hours and the state space increased exponentially to over 10^8.

When the first version of this CSP model of the OASIS NoC was given to FDR to verify for deadlock freedom, for even the 2×2 mesh, it took a hours to complete the check. The current model presented here was designed by imposing some restrictions on the network and removing those events that cannot occur. For instance since we know that not all routers will have all five input ports it was not necessary to include them in our model (although their inclusion would not have affected our results; it will only increase the number of states). Again we added the restriction that a router will not send a flit to itself and also paths are permitted for flits. These restrictions are what made us be able to produce this constrained version with less number of states which FDR was able to handle very well.

FIGURE 0-2 FDR Deadlock check of OASIS NOC



Chapter 7

Model Checking OASIS with PRISM

The PRISM model checker was designed to formally verify a probabilistic systems for probabilistic properties. In the chapter we show how PRISM can be used to verify properties which are not really probabilistic. As a comparison with FDR we use PRISM to verify our OASIS NoC for deadlock freedom. We also show the verification of an interesting property of NoCs, namely buffer property, using PRISM. Buffer property specification in CSP was treated in depth by Roscoe[] and therefore we did not consider it our treat it in this work when we were verifying OASIS in FDR.

The idea we used in this work to show how PRISM could be used to verify non probablistic properties is to specify the properties as if they were probabilistic.

The design of the model as well as properties verified is presented and finally the verification results also presented.

7.1 Prism model of OASIS

Our PRISM model of the OASIS NoC is a Markov Decision Processes (MDP) which represents a nondeterministic model of the system. We modeled the system as an MDP because we wanted the choice between actions (or commands) to be nondeterministic instead of the choice been uniformly random in which case our model would have been a DTMC.

As with our CSP model described in the previous section, here too we modeled a 2×2 mesh NoC. Each router was modeled as a PRISM module; so in all we had four modules representing the four routers.

Error! Reference source not found. shows a diagram of PRISM module of *Router_00* nd **Figure 0-2** shows the PRISM code for this module. The comments included in the PRISM code describe the module structure and how the module works.

Figure 0-1 Diagram of PRISM module of Router_00

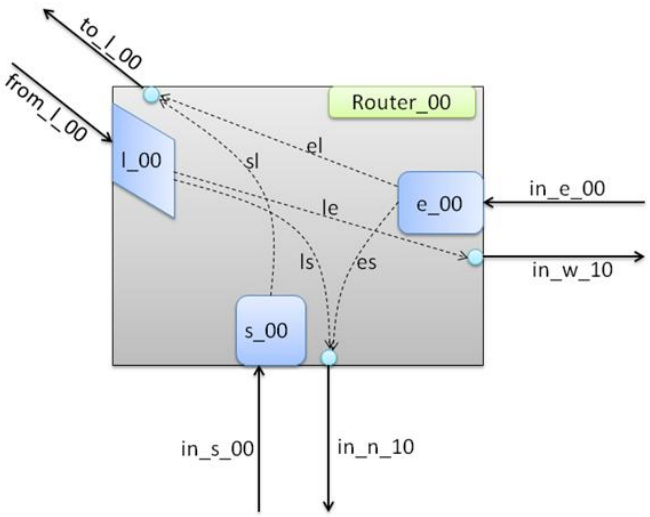


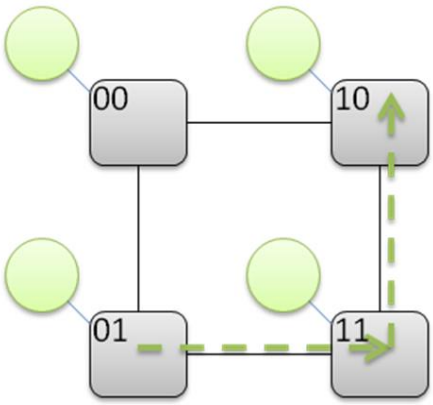
FIGURE 0-2 Router_00 PRISM module

```
const int f = 2; //input buffer size
const double p=0.5; //probability of a flit entering the NoC
const int N; //maximum number of flits that can enter NoC
global in_sys : [0..N]; //variable to keep track of number flits entering NoC - number of flits leaving NoC

module ROUTER_00
 l_00 : [0..f] init 0; //local input port
 s_00 : [0..f] init 0; //south input port
 e_00 : [0..f] init 0; // east input port
 le_00 : [0..f] init 0; //local to east
 ls_00 : [0..f] init 0; //local to south
 es_00 : [0..f] init 0; //east to south
 el_00 : [0..f] init 0; //east to local
 sl_00 : [0..f] init 0; //south to local

//with a probability of p, a flit enters the local input port from the local block.
//the flit has equal probabilty of leaving any of the possible output ports, i.e east or south
 [] l_00<f & le_00<f & ls_00<f & in_sys<N -> p/2:(l_00'=l_00+1)&(le_00'=le_00+1)&(in_sys'=in_sys+1)
                                            + p/2:(l_00'=l_00+1)&(ls_00'=ls_00+1)&(in_sys'=in_sys+1)
                                            + (1-p):true;

 [in_n_01] ls_00>0 & l_00>0 -> (ls_00'=ls_00-1) & (l_00'=l_00-1); //south output from local input
 [in_n_01] es_00>0 & e_00>0 -> (es_00'=es_00-1) & (e_00'=e_00-1); //south ouput from east input
 [in_s_00] s_00<f & sl_00<f -> (s_00'=s_00+1) & (sl_00'=sl_00+1); //south input

//east input. flit may be heading towards local or south output port
 [in_e_00] e_00<f & el_00<f & es_00<f -> 1/2:(e_00'=e_00+1)&(el_00'=el_00+1)
                                        + 1/2:(e_00'=e_00+1 ) & (es_00'=es_00+1);

 [in_w_10] le_00>0 & l_00>0 -> (le_00'=le_00-1) & (l_00'=l_00-1); // east output

//flit leaves local output to block
 [] el_00>0 & e_00>0 & in_sys>0 -> (el_00'=el_00-1)&(e_00'=e_00-1)&(in_sys'=in_sys-1);
 [] sl_00>0 & s_00>0 & in_sys>0 -> (sl_00'=sl_00-1)&(s_00'=s_00-1)&(in_sys'=in_sys-1);

endmodule
```

The entry of a flit from a local block into the local input port of a router is probabilistic with a value *p* as shown in the module definition of **Figure 0-2**. Once a flit enters an input there is an equal chance of it moving in any of the "allowed" output ports. Allowed in the sense that because of the XY routing algorithm used in the OASIS NoC some paths are not allowed as has been explained in previous chapters. For instance in our module definition for *Router_00* as shown in **Figure 0-2**, if a flit enters the *Router_00* through the South input port, it can only be output through the Local output port because if it was meant to move west to *Router_10*, in

would not have come through this *Router_00* in the first place. That is if a flit is to be transferred from *Router_01* to *Router_10*, it has to pass through *Router_11* and not *Router_00* because it has to move in X direction first as define by the XY routing algorithm used in OASIS NoC. This path is shown in **Error! Reference source not ound.** movement is shown.

Figure 0-3 Path for a flit from ROUTER_01 to ROUTER_10



The PRISM model of the OASIS 2×2 NoC has *136,048,896 states* and *2,086,083,072 transitions*, and it took PRISM 41 iterations using the *Jacobi Iterative* method to compute the reachable states.

7.2 PRISM Verification of OASIS – Results and Analysis

For our PRISM model of OASIS we were interested in verifying the following properties.

- The probability of the NoC becoming full and remaining full is 0. That is the NoC cannot deadlock
- There is a probability of 1 that no flit gets lost in the NoC. That is the NoC behaves as a buffer.

7.2.1 Verification of deadlock-freedom

For the first case of deadlock-freedom since PRISM does not provide an inbuilt deadlock checking functionality as in the case of FDR we had to formulate some PRISM property that could help us verify such a behaviour. In our chosen example of

a 2×2 mesh topology, the network will deadlock if eventually all the input buffers of all four routers get full and remains full. This is because no flit will be able to enter the network and no router will be able to send a flit since each one will be waiting for the other to free some space to allow for the incoming flit.

This can be related to the way CSP reasons about deadlock using the *failures* of a process. Recall in Chapter 4 we stated that a process *P* can deadlock if and only if after performing a trace *s* it refuses to peform any event offered to it by its environment, i.e. $(s, \Sigma) \in failures(P)$. In the same way in our PRISM model, if after performing some sequence of events, eventually the network gets full and remains full, it will not be able to perform any event. If this happens then we can say our NoC can deadlock.

To be able to specify a PRISM property for this behaviour, we defined a PRISM label which represents the situation in which all the input buffers in the network are full as

```
label "NoC_full" =  l_00=f & s_00=f & e_00=f
                  & l_01=f & n_01=f & e_01=f
                  & l_10=f & s_10=f & w_10=f
                  & l_11=f & n_11=f & w_11=f;
```

The value *f* represents the maximum size of the input buffer; we chose 2 for our example.

We can now define the PCTL formula as:

```
Pmin=? [F G ("NoC_full") ]
```

and

```
Pmax=? [F G ("NoC_full")]
```

for the minimum and maximum probability that "eventually all the buffers get full and remain full from that point". Model checking these properties with PRISM gave both the minimum and maximum probabilities as 0 (See Appendix A for verification

results as shown by PRISM). This means that it is not possible for the network to get full and remain full. From this result we can say our model of the OASIS NoC cannot deadlock.

Maybe using [*F G*] (eventually always) was too strict so we decided to relax the property a bit and instead just look at what happens *next* if the network gets full. So we defined new PRISM properties to find the minimum and maximum probabilities that if the network becomes full, it remains full at the next time step as

```
Pmin=? [ F ("NoC_full" &( X "NoC_full")) ]

Pmax=? [ F ("NoC_full" &( X "NoC_full")) ]
```

Again model checking using PRISM gave both the minimum and maximum probabilities as 0 (See Appendix A for verification results as shown by PRISM).

From these verification results we can conclude that our OASIS NoC model is deadlock-free.

7.2.2 Buffer property verification

Another interesting property of NoCs is that they behave as buffers; that is flits enter and leave without any losses. We therefore decided to verify this property on our PRISM model of the OASIS NoC. We defined this property as follows.

If our NoC is to behave as a buffer, then we expect that at any point in time the number of flits that have entered the NoC (*in*) should be equal to the number that exists in the NoC (*present*) plus those that have left the NoC (out). That is

$$in = present + out$$

$$\Rightarrow \quad in - out = present$$

Assuming an external observer stands outside tour NoC model with a counter such that anytime a flit enters the NoC the counter is increased by 1 and if a flit exists from the NoC the counter is decreased by 1. Then at anytime we expect that this counter value should be equal to the value of the sum of the number of flits in each of the input buffers of our network. This counter value is what we defined in our PRISM model as the variable *in_sys*. As can be seen in the code snippet below, anytime there is an entry into the Local input port this value is increased by one and when there is an output out of the Local output port it is decreased by 1.

```
[] l_00<f & le_00<f & ls_00<f & in_sys<N ->
p/2:(l_00'=l_00+1)&(le_00'=le_00+1)&(in_sys'=in_sys+1)
                                        +
p/2:(l_00'=l_00+1)&(ls_00'=ls_00+1)&(in_sys'=in_sys+1)
                                        + (1-p):true;
[] el_00>0 & e_00>0 &in_sys>0 -> (el_00'=el_00-1)&(e_00'=e_00-1) & (in_sys'=in_sys-1);
```

To calculate the number of flits in the network we defined a PRISM formula which sums the flits in all the input buffers in the network as

```
formula num_flits = l_00 + s_00 + e_00 +
                    l_01 + n_01 + e_01 +
                    l_10 + s_10 + w_10+
                    l_11 + n_11 + w_11 ;
```

With these two variables (*in_sys* and *num_flits*) defined, we now state our PCTL property formulas as

```
P>=1 [ G num_flits = in_sys ]

P<=1 [ G num_flits = in_sys ]
```

which states that there is a minimum and maximum probability of 1 that always the number of flits in the network is equal to the number of flits which have entered the system minus those that have exited from the system.

Both properties were verified to be **true** in all states of our PRISM model (See Appendix A).

From these verification results we can conclude that our OASIS NoC behaves as a message buffer.

Chapter 8

Conclusion

This project has shown how formal verification can be applied to a Network on Chip, based on a case study of the OASIS NoC. We were interested in verifying that the OASIS NoC was free from deadlock and also behaved as a message buffer. Both refinement model checking and probabilistic model checking were used for this case study.

The OASIS NoC was first modelled in the CSP language and then verified with the FDR model checker to check if it was free from deadlock. Because FDR has an inbuilt deadlock checking facility, no deadlock property was specified. The results from the verification showed that the network was deadlock free. One observation worth noting is that increasing the size of the network just by one row and column of routers caused an exponential increase in the state space and it took far more time for FDR to finish the check. While it took FDR just 7 seconds to verify the 2×2 network, it took it over 12 hours for a 3×3 on the same computer.

The OASIS NoC was also formalized in PRISM and we showed how PRISM could be used to verify properties which are not probabilistic. Our results showed that the OASIS NoC was free from deadlock. We were also able to use PRISM to verify that the OASIS NoC behaved as a message buffer; receiving and delivering flits without any losses. Although it was obvious that because of the flow control mechanism used in OASIS there was not going to be any flit losses, the significance of this is that it has shown that even non probabilistic properties could be verified with PRISM. The importance of this result is that if a system has both probabilistic and non-probabilistic properties, PRISM can be used to very all the properties.

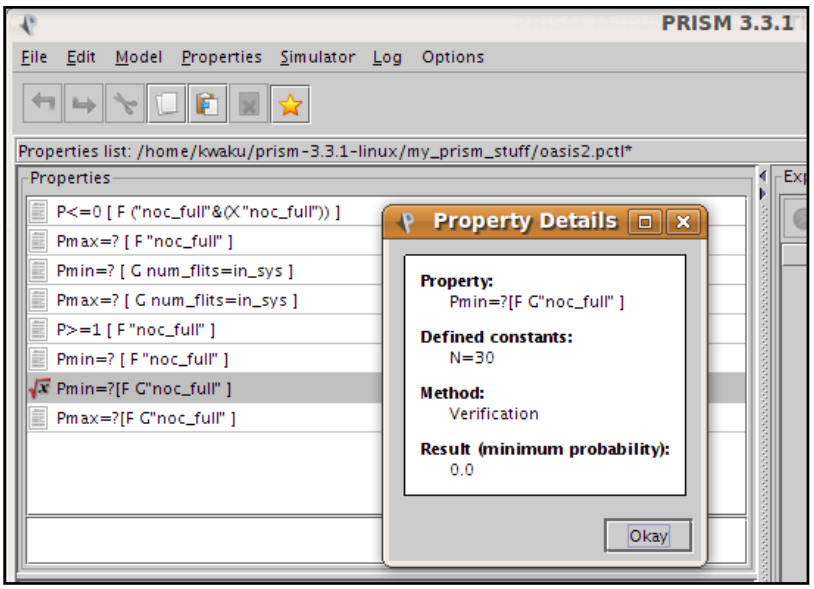
As can be seen from the model descriptions of both the CSP and PRISM, modelling the NoC to behave the way we wanted wasn't as difficult as trying to make a design which has a smaller state space.

Since NoCs can be characterized by some common set of properties such as the topology, switching technique, routing algorithm, buffering mechanism, control flow, etc, a future direction for this work will be to create a NoC modelling framework in CSP or PRISM. This metamodel would allow a designer to easily make a model of a NoC for verification in FDR or PRISM by simply instantiating this metamodel by specifying the properties of the NoC. From there a graphical interface could be created from this framework to allow for even easier modelling of NoCs in CSP or PRISM for verification in model checkers such as FDR and PRISM respectively.

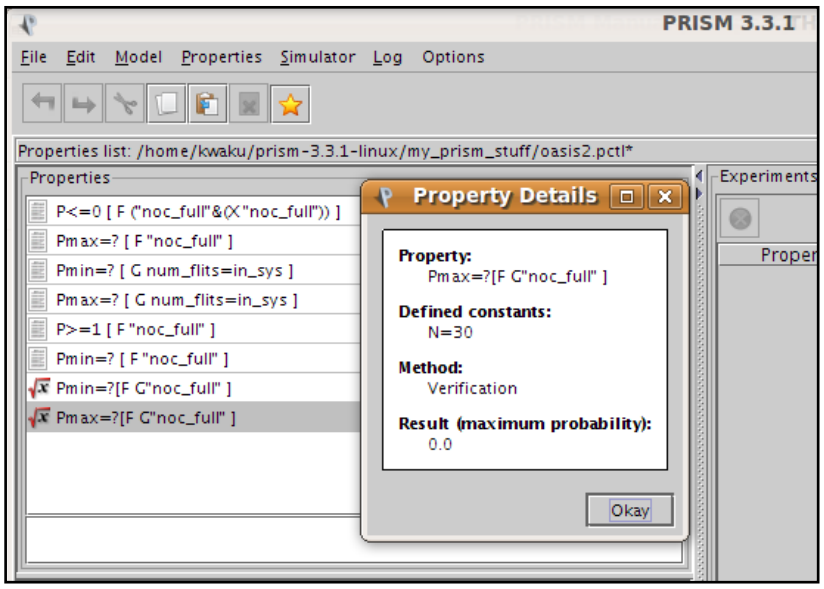
Appendix A

PRISM verification results

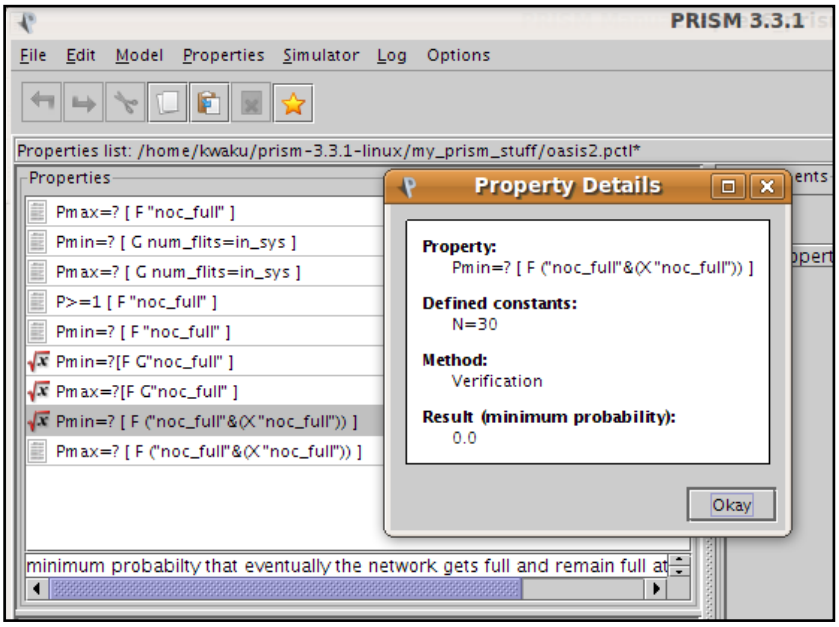
A.1 Minimum probability of NoC eventually getting full



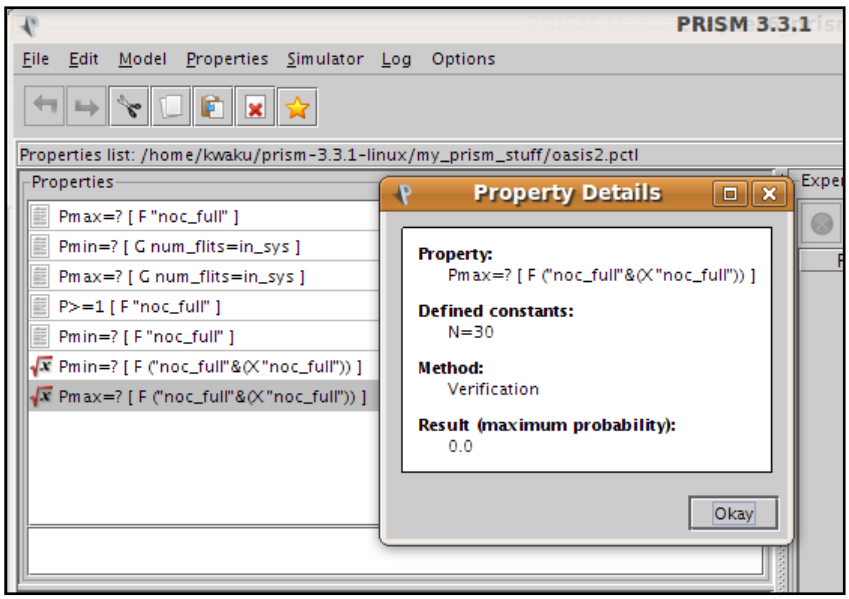
A.2 Maximum probability of NoC eventually getting full



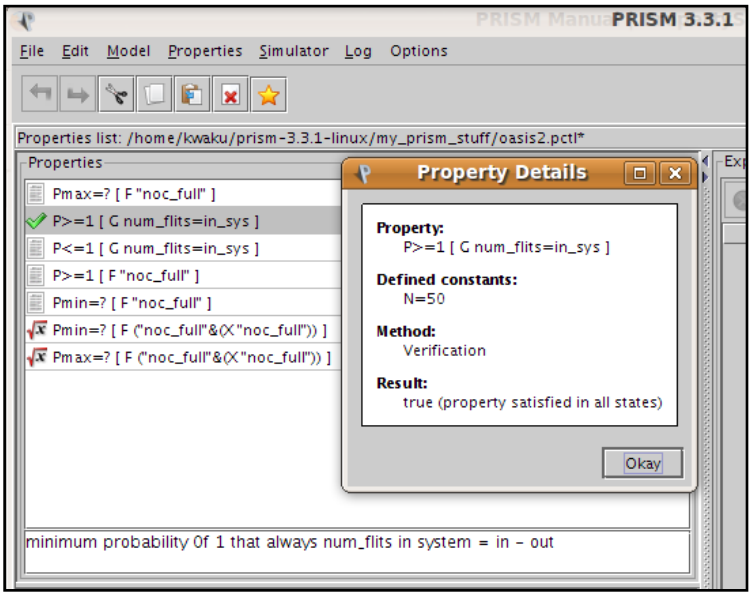
A.3 Minimum probability of NoC eventually getting full and still full at next instance



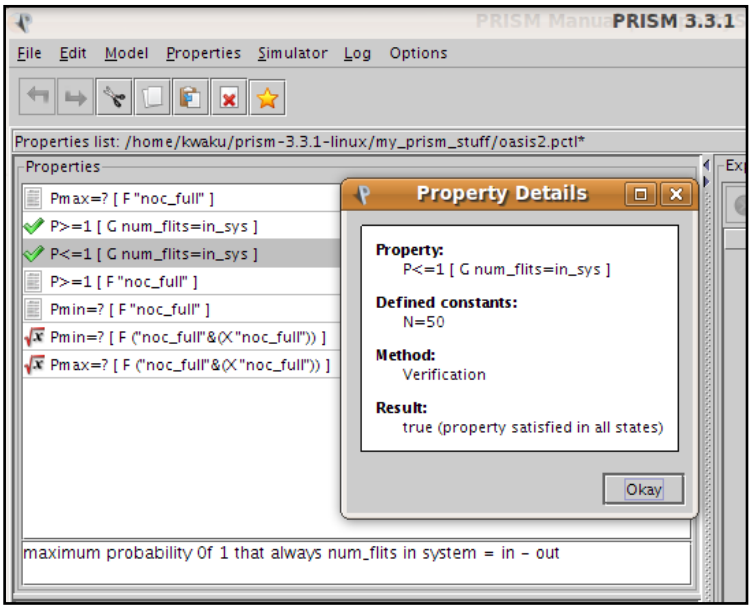
A.4 Minimum probability of NoC eventually getting full and still full at next instance



A.5 Minimum probability of 1 that NoC is a buffer



A.6 Maximum probability of 1 that NoC is a buffer



References

[AEK06] B. Ahmad, A. Erdogan, and S. Khawam, “Architecture of a dynamically reconfigurable NoC for adaptive reconfigurable MPSoC,” in AHS, Jun. 2006,pp. 405–411.

[AH96] R. Alur and T. Henzinger. Reactive modules. In Proc. 11th Annual IEEE Symposium on Logic in Computer Science (LICS), pages 207–218,1996.

[AS06] B. A. Abderazek and M. Sowa, “Basic Network-on-Chip Interconnection for Future Gigascale MCSoCs Applications: Communication and Computation Orthogonalization”, proceedings of the TJASSST '06 symposium on science, society and technology, Sousse, Dec. 4-6, 2006.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. IEEE Transactionson Computers, 35(8):677–691, 1986.

[BA95] A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. In Proc. FST & TCS, 1995.

[BB04] D. Bertozzi and L. Benini. Xpipes: a network-on-chip architecture for gigascale systems-on-chip. IEEE Circuits and Systems Magazine, 4(2):18–31,April 2004.

[BCGK04] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny. QNoC: QoS architecture and design process for network on chip. Journal of System Architecture, 50(2–3):105–128, 2004.

[BCM92] J. R. Burch, E. M. Clarke, and K. L. McMillan. Symbolic model checking: 10^{20} states and beyond. Information and Computation, 98:142–170, 1992.

[BHH00] C. Baier, B. Haverkort, H. Hermanns, and J. P. Katoen. Model checking continuous-time Markov chains by transient analysis. In CAV 2000, 2000.

[BHPS07] Dominique Borrione, Amr Helmy, Laurence Pierre, Julien Schmaltz, "A Generic Model for Formally Verifying NoC Communication Architectures: A Case Study," nocs, pp.127-136, First International Symposium on Networks-on-Chip (NOCS'07), 2007

[BK98] C. Baier and M. Kwiatkowska. Model checking for a probabilistic branching time logic with fairness. Distributed Computing, 11(3), 1998.

[BM06] T. Bjerregaard and S. Mahadevan, “A survey of research and practices of network-on-chip,” ACM Computing Surveys, vol. 38, no. 1, p. article No. 1,Jun. 2006.

[CE81] E.M. Clarke and E.A. Emerson, “Synthesis of synchronization skeletons for branching time temporal logic. In Logic of Programs: Workshop, Yorktown Heights, NY, May 1981, Volume 131 of Lecture Notes in Computer Science (1981). Springer-Verlag.

[CGP99] E. Clarke, O. Grumberg and D. Peled. Model Checking. The MIT Press, 1999.

[CMS95] R. Cleaveland, E. Madelaine and S. Sims, “Generating front ends for verification tools. In E. Brinksma, R. Cleaveland, K. Larsen, and B. Steffen Eds., Tools and Algorithms for the Construction and Analysis of Systems(TACAS ‘95), Volume 1019 of Lecture Notes in Computer Science(Aarhus, Denmark, May 1995), pp. 153-173. Springer-Verlag.

[CSC06] K.C. Chang, J.-S. Shen, and T.-F. Chen, “Evaluation and design trade-offs between circuit-switched and packet-switched NOCs for application-specific SOCs,” in DAC, Jul. 2006, pp. 143–148

[CW06] E. M. Clarke and J. M. Wing, “Formal Methods: State of the Art and Future Directions,” ACM Computing Surveys, Vol. 28, No. 4. (1996), pp. 626-643.Dec 1996

[Der70] C. Derman, Finite state Markovian decision processes. Academic Press,1970.

[DHPS07] D. Borrione, A. Helmy, L. Pierre, and J. Schmaltz. A Generic Model for Formally Verifying NoC Communication Architectures: A Case Study. In Proc. IEEE International Conference on NoCs (NoCs’07).

[DYN03] J. Duato, S. Yalamanchili, and L. Ni. Interconnection Networks - An Engineering Approach. Morgan Kaufmann, 2003.

[FDR05] FDR2 User Manual, June 2005, www.fsel.org

[FP07] B. Feero and P. Pande, “Performance evaluation for three-dimensional networks-on-chip,” in ISVLSI, 2007, pp. 305–310.

[FSEL] Formal Systems Europe Limited, www.fsel.org

[FV03] K. Fall and K. Varadhan, The ns Manual, pp.1-380, Dec.2003.

[GDR05] K. Goosens, J. Dielissen, and A. Radulescu. The aethereal network on chip: Concepts, architectures, and implementations. IEEE Design and Test of Computers, 22(5):21–31, September 2005.

[GG00] P. Guerrier and A. Greiner, “A generic architecture for on-chip packet-switched interconnections,” in DATE, Mar. 2000, pp. 250–256.

[GH04] Gerard J. Holzmann. The Spin Model Checker : Primer and Reference Manual., Addison-Wesley , 2004.

[GLM01] H. Garavel, F. Lang, and R. Mateescu. An Overview of CADP 2001. EASST Newsletter, 4:13–24, 2002.

[GVZ+05] B. Gebremichael, F. Vaandrager, M. Zhang, K. Goossens, E. Rijpkema, and A. Radulescu. Deadlock Prevention in the Æthereal protocol. In D. Borrione and W. Paul, editors, Correct Hardware Design and Verification Methods (CHARME'05), volume 3725 of LNCS, pages 345–348, 2005.

[Hoa04] C.A.R. Hoare, Communicating Sequential Processes, June 21, 2004

[Hol97] G. J. Holzmann. The Model Checker SPIN. IEEE Trans. on Soft. Eng., 23(5):279–295, 1997.

[HJ94] H. Hansson and B. Jonsson. A logic for reasoning about time and probability. Formal Aspects of Computing, 6(5):512–535, 1994.

[HN05] C. Hilton and B. Nelson, "A flexible circuit switched NOC for FPGA based systems," in FPL, Aug. 2005, pp. 24–26.

[HR04] M. Huth and M. Ryan. Logic in Computer Science - Modelling and Reasoning about Systems, Second Edition. Cambridge University Press, 2004.

[KND02] F. Karim, A. Nguyen, and S. Dey, "An interconnect architecture for networking systems on chips," IEEE Micro, vol. 22, no. 5, pp. 36–45, Sep.-Oct.2002.

[KNP02] M. Kwiatkowska, G. Norman, D. Parker,"PRISM: Probabilistic Symbolic Model Checker"T. Field, P. Harrison, J.Bradley and U. Harder (Eds.), Computer Performance Evaluation (TOOLS'02), volume 2324 of LNCS, pages 200–204, 2002.

[LST00] J. Liang, S. Swaminathan; R. Tessier," aSOC: A Scalable, Single-Chip communications Architecture. In: IEEE International Conference on Parallel Architectures and Compilation Techniques, Oct. 2000, pp. 37-46.

[Lin04] A. Lines, "Asynchronous interconnect for synchronous SoC design," IEEE Micro, vol. 24, no. 1, pp. 32–41, Jan.-Feb. 2004.

[McM93] K. L. McMillan. Symbolic Model Checking: An Approach to the State Explosion Problem. Kluwer Academic Publishers, 1993.

[MAK09] K. Mori, A. B. Abdallah, K. Kuroda , "Design and Evaluation of a Complexity Effective Network-on-Chip Architecture on FPGA The 19th Intelligent System Symposium (FAN 2009), Aizuwakamatsu, Sept. 2009

[Maz88] A. Mazurkiewicz. “Basic Notions of Trace Theory. In Workshop on Linear Time Branching Time, and Partial Order in Logics and Models for Concurrency, vol 354 of Lecture Notes in Computer Science, pp 285-363. Springer, 1988

[MCM+04] F. Moraes, N. Calazans, A. Mello, L. Miller, and L. Ost. “Hermes: an infrastructure for low area overhead packet-switching networks on chip,” The VLSI Journal Integration, 38(1):69–93, 2004.

[MNTJ04] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch, “Guaranteed bandwidth using looped containers in temporally disjoint networks within the nostrum network on chip,” in Design, Automation and Test in Europe Conference Proceedings, IEEE, February 2004.

[Nar04] P. Narain , Why full-chip formal verification is possible, EEdesign.com(11/08/2004)http://www.eetimes.com/news/design/columns/eda/showArticle.jhtml?articleID=52500185)

[NM93] L. M. Ni and P. K. McKinley. A survey of wormhole routing techniques in direct networks. IEEE Computer, 26(2):62–76, February 1993.

[NM93] L.M.Ni and P.K. McKinley, “A survery of Wormhole Routing Techniques in Direct Networks,” IEEE Comp. Mag., vol 26, no.2, Feb. 1993, pp. 62-76

[ORS92] S. Owre, J. M. Rushby, and N. Shankar,” PVS: A prototype veri cation system. In Deepak Kapur, editor, 11th International Conference on Automated Deduction (CADE), volume 607 of Lecture Notes in Arti cial Intelligence, pages 748-752, Saratoga, NY, June 1992. Springer-Verlag.

[Pel01] D. A. Peled, “Software Reliability Methods,” Springer-Verlag, ISBN: 0-387-95106-7, 2001.

[Pnu81] A. Pnueli, A temporal logic of concurrent programs. Theory of Computer Science 13, 45-60, 1981

[Pau94] L. C. Paulson. “Isabelle: A Generic Theorem Prover,” Springer, 1994.

[PRIS] PRISM Manual, www.prismmodelchecking.org

[QS82] J. Queille and J. Sifakis, Specification and verification of concurrent systems in CAESAR. In proc. Of Fifth ISP(1982)

[Ros94] A. Roscoe. Model-checking CSP. In Roscoe Ed. A Classical Mind: Essays in Honour of C. A. R. Hoare (1994), Prentice-Hall.

[Ros97] A.W. Roscoe, The Theory and Practice of Concurrency, Prentice Hall, ISBN: 0-13-674409-5.page , 1997

[Sch00] S. Schneider, “Concurrent and Real-time Systems: The CSP approach”, John Wiley and Sons Ltd, 2000, ISBN: 0-471-62373-3

[SALR05] D. Seo, A. Ali, W.T. Lim, and N. Rafique. Near-optimal worst-case throughput routing for two-dimensional mesh networks. In Proc. IEEE 32nd Int’l Symp. Computer Arch., pages 432–443, June 2005.

[SB05] J. Schmaltz, D. Borrione: "A Generic Network on Chip Model". Proc. of 18th International Conference on Theorem Proving in Higher order Logics, August 2005.

[SB07] J. Schmaltz and D. Borrione. A functional formalization of on chip communications. Formal Aspects of Computing, October 2007.

[SK04] H. Samuelsson and S. Kumar, “Ring road noc architecture,” in Norchip, 2004, pp. 16–19.

[SKH08] E. Salminen, A. Kulmala, and T. D. Hamalainen, “Survey of Network-on-chip Proposals ”, White paper, OCP-IP, Mar. 2008

[SSTV07] G. Salaün, W. Serwe, Y. Thonnart, and P. Vivet. "Formal Verification of CHP Specifications with CADP - Illustration on an Asynchronous Network-on-Chip". In Proceedings of the 13th IEEE International Symposium on Asynchronous Circuits and Systems ASYNC 2007 (Berkeley, California, USA), IEEE Computer Society Press, March 2007.

[STN02] D. Siguenza-Tortosa and J. Nurmi. Proteo: A new approach to Network-On-Chip. In Proc. IASTED Int’l Conf. on Communication Systems and Networks,Malaga, Spain, September, 2002.

[TF88] Y. Tamir and G.L. Frazier. High-performance multiqueue buffers for VLSI communication switches. In Proc. 15th Int’l Symp. Computer Architecture, pages 343–354, May-Jun 1988.

[Val98] A. Valmari, The State Explosion Problem, Lectures on Petri nets: advances in Petri nets: Springer-Verlang, Berlin-Heidelberg, 1998, 429–473

[W05] P. Wolkotte et al., “An energy-efficient reconfigurable circuit-switched network-on-chip,” in IPDPS, Apr. 2005, p. 155a.

[ZZHG05] H. Zimmer, S. Zink, T. Hollstein, and M. Glesner. Buffer-architecture exploration for routers in a hierarchical Network-on-Chip. In Proc. 19th IEEE Int’l Parallel and Distributed Processing Symp., page 171, Denver, CO, April, 2005.